Programming in Java

by
Willi-Hans Steeb
International School for Scientific Computing
email address of the author:

steebwilli@gmail.com

and
Yorick Hardy
International School for Scientific Computing
email address of the author:

yorickhardy@gmail.com

Web page:

http://issc.uj.ac.za

Contents

Preface

Besides C++ Java is now the most widely available and used object-oriented programming language. It is a very useful language that is successfully utilized by many programmers in many application areas. It is a reasonably carefully thought-out language where the design is based partly on acknowledged principles and partly on solid experience and feedback from actual use. Java is a powerful but lean object-oriented programming language. It makes it possible to program for the Internet by creating applets, i.e. programs that can be embedded in a web page. For example, an applet can be an animation with sound, an interactive game, or a ticker tape with constantly updated stock prices. However Java is more than a programming language for writing applets. It also can be used for writing standalone applications. It seems it is becoming the standard language for both general-purpose and Internet programming. Java is close to C++. It has taken many features of C++, but unfortunately discarded some of them, for example templates and multiple inheritence. To this lean core it has added garbage collection (automatic memory management), multithreading (the capacity for one program to do more than one task at the time), and security capabilities. Java is a platform consisting of three components: (1) the Java programming language, (2) the Java library of classes and interfaces (Java has a huge number of built-in classes and interfaces), and (3) the Java Virtual Machine.

One of the biggest advantages Java offers is that it is portable. An application written in Java will run on all platforms. Any computer with a Java-based browser can run the applications or applets written in the Java programming language. The Java Virtual Machine (JVM) is what gives Java its cross-platform capabilities. The Java file is not compiled into a machine language, which is different for each operating system and computer architecture, Java code is compiled into byte-code (platform independent).

The Java programming language is object-oriented, which makes program design focus on what we are dealing with rather than on how we are going to do something. Object-oriented languages use the paradigm of classes. A class is an abstract data type (ADT). A class includes both data and the methods (functions) to operate on that data. We can create an instance of class, also called an object, which will have all the data members and functionality of its class. The class paradigm allows one to encapsulate data so that specific data values or methods implementations cannot be seen by those using the class. Encapsulation makes it possible to make changes in code without breaking other programs that use that code. Java also includes inheritance, this means the ability to derive new classes from existing classes. The derived class, also called a subclass, inherits all the data and methods of the existing class, referred to as the parent class. A subclass can add new data members to those inherited from the parent class. With respect to methods the subclass can reuse the inherited methods as is, change them, and/or add its own new method.

Java includes a huge number of built-in classes and interfaces. The programmer can use already existing class as is, create subclasses to modify existing classes, or implement interfaces to augment the capabilities of classes.

In chapter 2 we give the basic concepts in Java. The given programs are very helpful for the beginners. They are also the building blocks for more complex applications. The widely used `String` class and the classes `StringTokenizer` and `StringBuffer` are introduced. Chapter 3 is devoted to classes and objects. Furthermore the wrapper classes, the container class `Vector`, the class `Math` for doing mathematics and the BigInteger and BigDecimal classes are introduced. The `Object` class is the ancestor of all classes and discussed in detail in section 3.6. Finally the `this` object is explained in detail. Chapter 4 deals with inheritance and abstract classes. The graphical user interface (GUI) is discussed in chapter 5 and a number of examples are provided. Chapter 6 introduces exception handling. File manipulations for reading from file and writing to files are introduced in chapter 7. Java is able to produce multi-threaded applications, which often form a part of applications including animations. Threads are discussed in chapter 8 and application of threads in animation are given in chapter 9. An introduction into networking together with a number of programs is given in chapter 10. Chapters 11 and 12 deal with the additions to Java for the version 1.2. Finally chapter 13 lists important Web sites for Java, JavaScript and HTML.

The level of presentation is such that one can study the subject early on in ones education in programming. There is a balance between practical programming and the underlying language. The book is ideally suited for use in lectures on Java and object-oriented programming. The beginner will also benefit from the book. The reference list gives a collection of textbooks useful in the study of the computer language Java. There are a number of good textbooks for Java available [1], [2]. For applications of Java in science we refer to W.-H. Steeb et al [6] and Steeb [5]. Comprehensive introductions into JavaScript are given by [3] and [4].

Without doubt, this book can be extended. If you have comments or suggestions, we would be pleased to have them. The email addresses of the author are:

whsteeb@uj.ac.za
steebwilli@gmail.com

The web site of the authors is

http://issc.uj.ac.za

Chapter 1

Introduction

1.1 Why Java ?

Java is a platform-independent object-oriented, multi-threading dynamically-linked programming language. Java was developed by Sun MicroSystems primarily for the use on the World-Wide-Web. It has a lot in common with C++. With Java we can produce three distinct types of programs, applets, applications, and beans.

Firstly, Java supports the central concepts of object-orientated programming: encapsulation, inheritance and polymorphism (including dynamic binding). It has good support for dynamic memory management and supports both, procedural and object-orientated programming.

However, other well-designed programming languages have failed against relatively poor competitors. Being a good programming language is not sufficient for survival. An additional important requirement for a powerful programming language is portability. If a firm replaces one computer system with another, only a minimal amount of recording (if any) should be required. Java compilers are available for virtually all machines and a high level of compatibility is ensured by the standard for Java.

All these points have contributed to Java being the fastest growing computer language for nearly all computer and operating systems and for nearly all software applications, ranging from scientific to administrative programs to real-time industrial applications and computer games.

1.2 Aim of Object-Oriented Programming

Object-oriented programming is the most dramatic innovation in software development in the last two decades. It offers a new and powerful way to cope with the complexity of programs. Using object-oriented programming we can write clearer, more reliable, more easily maintained programs. Integrating data and functions (methods) is the central idea of object-oriented programming.

The class is the foundation of Java's support for object-oriented programming, and is at the core of many of its advanced features. The class provides the mechanism by which objects are created. Thus a class defines a new data type, which can be used to create objects. A class is created by using the keyword `class`.

Object-oriented programming methods aim to achieve the following:

- To simplify the design and implementation of complex programs.
- To make it easier for teams of designers and programmers to work on a single software project.
- To enable a high degree of reusability of designs and of software codes.
- To decrease the cost of software maintenance.

In order to achieve these aims, object-oriented programming languages are expected to support a number of features:

1. Information hiding (encapsulation)
2. Polymorphism
3. Inheritance

Each of these concepts is discussed below.

1.2.1 Information Hiding

Information hiding is achieved by restricting the access of the user to the underlying data structures to the predefined methods for that class. This shifts the responsibility of handling the data fields correctly from the user of the code to the supplier. For example, a programmer using a Date class is usually not interested in the implementation details. These include the underlying data structure of Date (i.e. whether the month is stored as an integer, a string or an enumeration type) and the underlying code (i.e. how two dates are subtracted from each other). If, at a later stage a more efficient way of storing the date or of calculating the number of days between two dates is introduced, this should not affect the programmers who have been using the date class. They should not have to search through all their programs for any occurrence of Date and to make the relevant changes. Such a maintenance nightmare and the very high cost accompanying it is prevented by information hiding where access to the underlying data structures is given only via the predefined methods of the class. A further advantage of information hiding is that it can be used to guarantee the integrity of the data (e.g. to prevent the user from setting the month equal to 13). Java and C++ provide the keywords `private`, `protected` and `public` for data hiding.

1.2.2 Inheritance

Often one object shares a number of characteristics with another, but has a few additional attributes. For example, our firm might have a database for vehicles. For each vehicle the registration number, model, year and maintenance history are stored. Suppose the firm wants to expand this data base to include its trucks. However, they want to store additional information for trucks, such as the the payload and the number of axes (for toll-road purposes). Instead of rewriting all the code for trucks, one can see a truck as a special case of a vehicle having all the attributes of a vehicle plus a few additional attributes. By making truck a derived class of vehicle, it inherits all the attributes of vehicle and the programmer has only to add the code for the additional attributes. This not only reduces the amount of coding to be done and maintained, but it also ensures a higher level of consistency. Should the country decide to change its format for number plates, it has to be only changed in the base class vehicle and the trucks automatically inherit the change. In Java a superclass is the parent of a class. The keyword is `super`. It is the class from which the current class inherits. In Java, a class can have only one superclass. Thus Java is a single inheritance system, in which a class can only inhert from one class. C++ is a multiple inheritance system in which a class can inherit from one or more classes.

1.2.3 Polymorphism

Often one wants to perform an action such as editing on a number of different objects (i.e. text files, pixel files, charts, etc.). Polymorphism allows one to specify an abstract operation like editing, leaving the actual way in which this operation is performed to a later stage. In the case where dynamic binding (linking during run-time) is used the decision of which code to be used for the editing operation is only made during run-time. This is especially useful in the case where the actual type of object which is to be edited is only known at run-time.

Furthermore, if a new object type is to be supported (e.g. faxes) then the new editing code (provided by the fax-software supplier) can be linked in at run-time. Even when the original program is not recompiled, it can support future additions to the system. Polymorphism is hence again a higher level of abstraction, allowing the programmer to specify an abstract action to be performed on abstract objects. Java allows defining abstract classes.

1.2.4 Built-In Classes

Java code is organized into classes (abstract data types). Classes define a set of *methods* (member functions) and *fields* (data members) that form the behaviour of an object. What makes Java so powerful is the large number of built-in classes. Classes are grouped by directories on disk into categories of related classes called packages. However, from version 1.0 to version 1.1 a large number of methods have been *deprecated* and replaced by other methods. Java 1.2 added a large number of new classes.

Some of the most important built-in classes are

`String, System, Graphics, Object, Vector, Math, Date, Event`

The `String` class does all the string manipulations such as `length()` to find the length of a string and `charAt(int)` which finds the character at the given position. The `System` class includes the method `exit()`. The `Graphics` class includes all drawing methods, for example `drawLine(int,int,int,int)` which draws a line between to points given by the arguments. The class `Object` is the mother of all classes. The `Vector` is a container class. Mathematical operations are in the class `Math`.

We use `import` statements to load classes required to compile a Java program. For example, to tell the compiler to load the `Applet` class from the `java.applet` package, the statement

```
import java.applet.Applet;
```

is used.

1.2.5 Java Compared To C++

The syntax of Java looks very similar to C++. However, there are number of important differences.

Java compared to C++ has

no pointers,

no references,

no structs,

no unions,

no typedefs,

no `#define`,

no need to manage/free (delete) memory,

no core dumps,

no `goto` although Java has a keyword `goto`,

no `sizeof` operator.

The size of the basic data types in Java are fixed. C++ has signed and unsigned data types for `short`, `int` and `long`. Java has no unsigned data basic data type. All integer basic data types are signed. There are no standalone methods in Java. Everything in Java must be in a class even the `main` method. In Java basic data types can only be passed by value. Arrays are passed by reference. Operators such as `+`, `*` cannot be overloaded in Java. Java does not allow function or class templates. Java has a boolean data type called `boolean`. Only newer C++ compilers have a boolean data type called `bool`. There is no multiple inheritance in Java. There is no scope resolution operator `::` in Java. Java uses the dot `.` (access operator) for everything. There are no global functions or global data in Java. If we want the equivalent of globals, we use `static` methods and `static` data within a class. Java does not have the C++ concept of a destructor, a method that is automatically called when an object is destroyed. The reason is that in Java the practice is simply to forget about objects rather than to destroy them, allowing the garbage collector to reclaim the memory as necessary. Java has both kinds of comments like C++ does.

1.3 Identifiers and Keywords in Java

Variables, constants, classes, objects, methods and interfaces are all identified by their names. A valid Java identifier must start either with a lower or upper case letter, an underscore character `(_)` or a dollar sign `($)`. Subsequent characters can also include numerals (0-9). The length of the identifier is not limited. Furthermore Java is case sensitive. The only restriction is that we may not use any of Java's reserved words. The following identifiers are thus all valid and all distinct

```
x X _x x1 myClass
```

while the following are not valid identifier names in Java

```
x!  1X  true  #x1
```

The identifier `true` is invalid because it is a reserved word. Since Java is case sensitive we could use `True` as an identifier.

Java's reserved words (so-called keywords) are

```
abstract  boolean       break      byte
byvalue   case          catch      char
class     const         continue   default
do        double        else       extends
false     final         finally    float
for       goto          if         implements
import    instanceof    int        interface
long      native        new        null
package   private       protected  public
return    short         static     super
switch    synchronized  this       threadsafe
throw     transient     true       try
void      while
```

where

```
byte, short, int, long, float, double, char, boolean
```

describe the basic (primitive) data types. The keyword `class` indicates an abstract data type, for example

```
class Rational
```

Chapter 2

Java Basics

2.1 My First Java Program

The following Java program displays the string `Good Morning Egoli` on the screen. The assumption is that the JDK (Java Developer Kit) is being used. The instructions for compiling and running are specific to the JDK.

```
// First.java (file name)

public class First
{
   public static void main(String[] args)
   {
   System.out.println("Good Morning Egoli");
   }
}
```

The file name is `First.java`. The extension of a Java program file is `.java`. To compile the code from the command line we enter

```
javac First.java
```

This produces a Java byte-code file called

```
First.class
```

To run the program from the command line we enter

```
java First
```

This command is also case-sensitive. Both `javac` and `java` are execute files in the `bin` directory.

In a Java program the name of the file with extension `java` must be the same as the name of the class which contains the `main` method. In the program given above the file name is `First.java` and the class name which contains `main` is `First`. Java is case sensitive. The keyword `class` is there because everything in a Java program lives inside a `class`. Following the keyword `class` is the name of the class. The compiled byte code is then automatically called `First.class` (platform independent). We have declared the class `First` `public` so that it is publicly accessible.

In Java, as in C/C++, curly braces

```
{ ... }
```

are used to delineate the parts (usually called blocks) in the program. A Pascal programmmer can relate them to the begin/end pair.

Every Java program must have a `main` method within a class for our code to compile. The method `main` is declared `static`. A static member method of a class can be invoked independently of any class instance. Instances of classes are called *objects*. A public class is defined within its own file and is visible everywhere. The `main` method must also be declared `public`. The `main` method returns `void` which is Java's way for specifying nothing. The `main` method has as argument an array of strings. Arrays are denoted in Java by a pair of square brackets `[]`. The name of the array of strings is `args`. This array will hold any command line parameters passed to our application.

Comments are the same as in C and C++, i.e.

```
// ....
/* ... */
```

The `System` class contains several useful class fields and methods. The method

```
static PrintStream out
```

is the standard output stream (normally to the screen). The method

```
void println(String s)
```

is in class `PrintStream` and prints a string and terminates the line. The method

```
void print(String s)
```

prints a string without a newline.

2.2 My First Applet

An *applet* is a Java program that must be run from another program called its host program. For example, applets are usually run from Webbrowsers such as Netscape's Communicator or Microsoft Internet Explorer. To run the applet from the webbrowser or using the `appletviewer` command provided in the Java Developer Kit (JDK) we need an `HTML` file. We show now how to run an applet from the command line. The program is

```
// Welcome.java

import java.applet.Applet;
import java.awt.Graphics;

public class Welcome extends Applet
{
   public void paint(Graphics g) // g is an object of class Graphics
   {
   g.drawString("Welcome to JAVA",25,25);
   g.drawString("Welcome to ISSC",50,50);
   }
}
```

To run the applet do the following. We compile `Welcome.java` by entering the follwing command at the command line:

```
javac Welcome.java
```

This creates a file

```
Welcome.class
```

Then we write the following file with file name `Welcome.html`

```
   <HTML>
   <COMMENT> file name: Welcome.html </COMMENT>
   <APPLET CODE="Welcome.class" width=275 height=135>
   </APPLET>
   </HTML>
```

3. Then at the command line enter

```
appletviewer Welcome.html
```

The command is provided with Java's Developer's Kit. This means `appletviewer` is an execute-file in the directory `jdk1.4\bin`. The program displays the messages

```
Welcome to JAVA
Welcome to ISSC
```

on the screen at the coordinates 25 and 25 and 50 and 50, respectively. Coordinates are measured from the upper-left corner of the applet in pixels. A pixel (picture element) is the unit of display for the computer screen. Many computers have 640 pixels for the width of the screen and 480 pixels for the height of the screen.

The program contains the definition for the method (member function)

```
public void paint(Graphics g)
```

This method is in the class `Component`. The `paint` method is called automatically during an applet's execution and is used to display information on the screen. The `paint` method's parameter list indicates that it requires a `Graphics` object (that is named `g` in our program) to perform its task. The keyword `public` is required so the browser can automatically call the `paint` method.

The left brace `{` begins the method definition's body. A corresponding right brace `}` must end the method definition's body.

The line

```
g.drawString("Welcome to JAVA",25,25);
```

instructs the computer to perform an action, namely to use the method

```
drawString(String str,int x,int y)
```

to draw the text given by the specified string using this graphics context's current font and color. Thus the method `drawString` is provided in the `Graphics` class.

The *scope* of an identifier is the portion in which the identifier can be referenced. For example, when we declare a local variable in a block, it can be referenced only in that block or in the blocks nested within that block. The scopes for an identifier are class scope and block scope. Block begins are indicated by the left brace `{` and the block end by the right brace `}`. An exception from this rule are `static` methods. In a sense, all instance variables and methods of a class are global to the methods of the class in which they are defined, i.e. the methods can modify the instance variables directly and invoke other methods of the class.

2.3 Basic Data Types

The *basic data types* (also called *primitive data types*) in Java for integers are

```
byte, short, int, long
```

```
type storage requirement range (inclusive)
byte  1 byte = 8 bits    -128 to 127
short 2 bytes = 16 bits  -32768 to 32767
int   4 bytes = 32 bits  -2147483648 to 2147483647
long  8 bytes = 64 bits  -9223372036854775808L to 9223372036854775807L
```

There is no unsigned data type in Java. This is problem when we consider IP addresses. IP addresses are arrays of unsigned byte. There is no `sizeof()` operator in Java. In C and C++ one has a `sizeof()` operator. In Java the size of the basic data types are fixed. The basic data types `byte`, `short`, `int`, `long` are initialized to 0. Default integer data type is `int`. We can declare a variable at any point in the program. The variable exists from the point of declaration until the end of the block in which it is defined.

The operator

```
(type)
```

is called the *type conversion operator*, for example `(int)`. An example is

```
byte a = (byte) 7;
int i = (int) a;
```

What happens if we change the line:

```
byte l = (byte)(j + k);
```

to

```
byte l = j + k;
```

in the following program? What happens if we change the line

```
c = (short)(a + b);
```

to

```
c = a + b;
```

in the following program?

```
// Types1.java
// Arithmetic operations:
// addition +, subtraction -

public class Types1
{
   public static void main(String[] args)
   {
   byte z1 = (byte) 255;
   System.out.println("z1 = " + z1); // => -1 since 255-256 = -1
   byte z2 = (byte) 152;
   System.out.println("z2 = " + z2); // => -104 since 152-256 = -104
   byte z3 = (byte) 1001;
   // repeated subtraction of 256 until
   // number is in range -128 ... 127
   System.out.println("z3 = " + z3); // => -23

   byte j = (byte) -128;
   byte k = (byte) 127;
   byte l = (byte)(j + k);
   System.out.println(l);    // => -1

   short a = (short) 8;
   short b = (short) -17;
   short c;
   c = (short)(a + b);
   System.out.println(c);   // => -9

   int d = -2000000;
   int e = 2000001;
   int f = d + e;
   System.out.println(f);  // => 1

   byte g = -5;
   int h = (int) g;
   System.out.println(h);  // => -5

   long p = 900000000000002L;
   long q = 100000000000003L;
   long s = p - q;
   System.out.println(s);  // => 799999999999999
   }
}
```

The floating point data members are `float` and `double`.

```
type storage requirement   range (inclusive)
float  4 bytes = 32 bits    pm 3.40282347E+38F
double 8 bytes = 64 bits    pm 1.79769313486231570E+308
pm stands for plus-minus
```

The default floating point data type is `double`. There is no `long double` (80 bits) in Java. All floating-point types follow the IEEE 754 specification. They will overflow on range errors and underflow on operations like a divide by zero. The floating data types are initialized to `0.0`. Java allows us to make certain assignment conversions by assigning the value of a variable without an explicit cast. Those that are permitted are

```
byte -> short -> int -> long -> float -> double
```

```
// Types2.java

public class Types2
{
   public static void main(String[] args)
   {
   float z1 = (float) 8.17;   // type conversion
   float z2 = (float) -912;   // type conversion
   float z3;
   z3 = z1 + z2;
   System.out.println(z3); // => -903.83

   double u = 20.1E+10;
   double v = 17.157;
   double w = u*v;
   System.out.println(w); // => 3.448557e+12

   // type conversion
   double x = -10.345;
   int nx = (int) x;        // type conversion
   System.out.println(nx);  // => -10

   int i = 10;
   int j = 3;
   double y = 10/3;   // integer division on right-hand side,
                      // then type conversion to double
   System.out.println(y);   // => 3.0
   double b = ((double) i)/((double) j);
   System.out.println(b);   // => 3.3333333333333335
   }
}
```

The `char` type uses single quotes to denote a *character*. For example

```
char ch = 'A';
```

The size of of data type `char` is 2 bytes compared to 1 byte for a character in C and C++. The `char` type denotes character in the *Unicode* encoding scheme. Unicode was designed to handle essentially all characters in all written languages in the world. It is a 2-byte code. This allows 65536 characters, unlike ASCII/ANSI, which is a 1-byte code allowing only 255 characters. The ASCII/ANSI code is a subset of Unicode. It is the first 255 characters on the Unicode coding scheme. For example

```
int i = (int) 'A';  // => 65
```

The `boolean` data type has two values: `false` and `true`. It is used for logical testing using the relational operators that Java, like C and C++, supports. Type conversion from `boolean` to `byte`, `short`, `int`, `long` is not possible. The storage requirement for the boolean data type is 1 byte.

Basic data types in Java, C, and C++ are compared for equality using the *equality operator*

```
==
```

For example

```
char c1 = 'X';
char c2 = 'x';
boolean b = (c1 == c2);
System.out.println(b);   // => false (Java is case sensitive)
```

A *string* is an array of characters. The data type `String` is an abstract data type in Java. Strings in Java are immutable. They cannot be changed. We have two methods to create a string, namely

```
String s1 = "abc";
String s2 = new String("abc");
```

The `equals` method provided in the `String` class compares two strings (case sensitive) for equality. The method returns `true` if the objects are equal and `false` otherwise. The method `equals` uses a lexicographical comparsion the integer Unicode values that represent each character in each String are compared. Thus the method `equals` is case sensitive.

```
// Types3.java

public class Types3
{
   public static void main(String[] args)
   {
   int r1;
   char ch1 = 'Y';  char ch2 = 'Z';
   if(ch1 != ch2)  r1 = 0;
   else  r1 = 1;
   System.out.println(r1);  // => 0

   int r2;
   char ch3 = 'A';  char ch4 = 'A';
   if(ch3 == ch4)  r2 = 0;
   else  r2 = 1;
   System.out.println(r2);  // => 0

   int x = 7;  int y = 8;
   boolean bool = true;
   if(x == y) System.out.println(bool);
   else
   System.out.println(!bool);  // => false

   String c1 = "egoli";   // String is an abstract data type
   String c2 = "egoli";   // Java provides a String class
   boolean b1 = c1.equals(c2);
   System.out.println(b1);        // => true

   String d1 = "Cape";
   String d2 = "cup";
   boolean b2 = d1.equals(d2);
   System.out.println(b2);       // => false

   String e1 = new String("Y");
   String e2 = new String("y");
   boolean b3 = e1.equals(e2);
   System.out.println(b3);      // => false
   }
}
```

2.4 Arithmetic Operations

The arithmetic operators are:

`+, -, *, /, %, ++, --`

where `%` is the *remainder operator* (also called *modulus operator*). For example `23%7 = 2`, since `23/7 = 3` in integer division and the remainder is 2. The operators `++` and `--` are the increment and decrement operators, respectively. We have a preincrement operator `++i` and a postincrement operator `i++` and analogously for the decrement operator. The combined operators are

`+=, -=, *=, /=, %=`

Thus, for example,

```
a += b;
```

is equivalent to

```
a = a + b;
```

The support for constants in Java is not as mature as that provided by C++. In Java one can define constant data members of a class. The keyword used for constants is `final`. For example

```
final double g = 1.618;
```

```
// Arith.java

public class Arith
{
    public static void main(String[] args)
    {
    int a = 7;
    int b = 8;
    int c = a + b;
    System.out.println(c); // => 15

    int d = 23;
    int e = -25;
    int f = d - e;
    System.out.println(f); // => 48

    int dd = 5;
    dd++;        // increment operator
    System.out.println(dd); // => 6
```

```
    int xx = -17;
    xx--;        // decrement operator
    System.out.println(xx); // => -18

    short k = (short) 5;
    short j = (short) 7;
    short l = (short) (k*j);
    System.out.println(l); // => 35

    long n = 3000000001L;
    long m = 2L;
    long result = n/m;     // integer division
    System.out.println(result); // => 1500000000

    long remain = n%m;
    System.out.println(remain);  // => 1 remainder

    int x = 7;
    x += -9;                 // short cut for x = x - 9;
    System.out.println(x);  // => -2

    double z1 = 1.0;
    double z2 = 3.0;
    // floating point division
    double z3 = z1/z2;
    System.out.println("z3 = " + z3); // => 0.333333...

    double d1 = 3.14159;
    double d2 = 2.1;
    d1 /= d2;        // short cut for d1 = d1/d2;
    System.out.println(d1);  // => 1.4959952...

    int u = 27;
    int v = 12;
    u %= v;                 // short cut for u = u%v;
    System.out.println(u); // => 3
    }
}
```

2.5 Unicode, ASCII Table, and Types Conversion

Unicode is the international standard character set that Java uses for its `String` and `StringBuffer` classes. Each code is a 16-bit integer with unique value in the range 0 to 65 535. These values are usually expressed in hexadecimal form. For example, the *infinity symbol* ∞ has the Unicode value 8734, which is 221E in hexadecimal. In Java, the character whose Unicode is `hhhh` in hexadecimal is expressed as `\uhhhh`. For example, the infinity symbol ∞ is expressed as `\u221E`. The first 127 values are the same as the ASCII Code (American Standard Code for Information Interchange). The following program shows the conversion from `char` to `int` and `int` to `char` where the ASCII table is taken into account.

```
// Unicode.java

public class Unicode
{
   public static void main(String[] args)
   {
   char c1 = (char) 65;    // type conversion ASCII table
   System.out.println(c1); // => 'A'

   char c2 = (char) 97;
   System.out.println(c2); // => 'a'

   char a = (char) 3;
   System.out.println(a);

   int i = (int) 'A';
   System.out.println(i); // => 65

   int j = (int) 'a';
   System.out.println(j); // => 97

   int k = (int) '0';
   System.out.println(k); // => 48

   char n = '\0';          // null character
   System.out.println("n = " + n); // no output

   char inf = '\u221E';
   System.out.println("inf = " + inf); // => ? (explain why)
   }
}
```

In the following program we show how to display degree Fahrenheit, degree Celsius, infinity (`hex 221E`) and the Euro (`hex 20AC`).

```
// UniCode.java

import java.awt.*;
import java.awt.event.*;

class UniCode extends Frame
{
    Label label1, label2, label3, label4;

    UniCode(String s)
    {
    super(s);
    setSize(200,200);
    addWindowListener(new WindowAdapter()
    { public void windowClosing(WindowEvent event)
    { System.exit(0); }});

    setLayout(new FlowLayout());
    label1 = new Label("             ");
    label1.setFont(new Font("TimesRoman 12 point bold.",20,20));
    add(label1);
    label2 = new Label("             ");
    label2.setFont(new Font("TimesRoman 12 point bold.",20,20));
    add(label2);
    label3 = new Label("             ");
    label3.setFont(new Font("TimesRoman 12 point bold.",20,20));
    add(label3);
    label4 = new Label("             ");
    label4.setFont(new Font("TimesRoman 12 point bold.",20,20));
    add(label4);
    label1.setText("\u00B0F");
    label2.setText("\u00B0C");
    label3.setText("\u221E");
    label4.setText("\u20AC");
    setVisible(true);
    }

    public static void main(String[] args)
    {
    new UniCode("Example");
    }
}
```

2.6 Precedence Table

Precedence is the priority system for grouping different types of operators with their operands. Java, C and C++ have precedence rules that determine the order of evaluation in expressions that contain more than one operator. When two operators have unequal precedence the operator with higher precedence is evaluated first. Operators with equal precedence are evaluated from left to right. For example, the fourth line of the code fragment

```
int a = 5;
int b = 6;
int c = -1;
int r = a*b + c;   // => 29
int s = a*(b + c); // => 25
```

contains three operations, namely `=, *, +`. After the precedence table the multiplication operators is applied first, then the addition operator and finally the assignment. Thus `r` takes the value 29. In the expression `a*(b+c)` we first evaluate `b+c` and then multiply with `a`. Thus the output for `s` is 25.

The operator `++` is the increment operator, it increments by 1. We have to keep in mind that there is a prefix increment and a postfix increment. They are at different places at the precedence table. A lot of confusion in literature is about the increment and decrement operators, `++` and `--`, respectively. Let `x` be an integer. There is a *preincrement operator* `++x` and a *postincrement operator* `x++`. The preincrement operator is at the top of the precedence table and postincrement operator is at the bottom of precedence table. Notice that the postincrement operator is even below the assignment operator `=`. The same applies for the decrement operator `--`.

The access operator `.` is higher in the precedence table than the multiplication operator.

The operators `*` (multiply), `/` (division) and `%` (remainder) have equal precedence. Operators with equal precedence are evaluated from left to right. All three operators have higher precedence than the assignment operator `=`. For example

```
int m = 27;
int n = 8;
int p = 5;

int r1 = m/n%p;  // => 3
int r2 = m%n/p;  // => 0
```

C and C++ follow the same rules for these operators.

Fill in the output for the following program. Then run the program and compare.

```
public class Preced
{
   public static void main(String[] args)
   {
   int a = 7;
   a++;
   ++a;
   System.out.println(a); // =>

   int b = 5, c = 7;
   int r1 = c*b++;
   System.out.println(r1); // =>

   System.out.println(b);  // =>

   int d = 5, e = 7;
   int r2 = d*(e++);
   System.out.println(r2); // =>

   int f = 5, g = 7;
   int r3 = f*++g;
   System.out.println(r3); // =>

   int n = 23;
   int m = 5;

   int s1 = n/m;
   System.out.println(s1); // =>

   int s2 = n%m;
   System.out.println(s2); // =>

   int p = 3;
   int s3 = n/m%p;
   System.out.println(s3); // =>

   int i = 12;
   int j = 12;
   System.out.println(i++); // =>
   System.out.println(++j); // =>
   }
}
```

2.7 Arrays

An *array* is a set of elements with the same data type. The first subscript of a Java, C and C++ array is 0 and the last subscript is 1 less than the number used to declare the array. Thus an array is a contiguous region of storage, large enough to hold all its elements. The elements in the array are accessed using the rectangular bracket `[]`.

An array can be declared and initialized in a single statement. For example

```
int vec[] = { 1, 78, 12, 5 };
```

allocates memory for four integer variables, initializes these memory positions for the integer values. The length of an array of basic data types or abstract data types can be found with the `length` command. For example

```
int l = vec.length;
System.out.println(l); // => 4
```

We can also use the `new` operator to allocate memory for the array and then assign the values. Any valid data type can be allocated using `new`. For example

```
int vec[] = new int[4];
vec[0] = 1; vec[1] = 78; vec[2] = 12; vec[3] = 5;
```

Java provides automatic memory management. It allocates a portion of memory as and when required. When memory is short, it looks for areas which are no longer referenced. These areas of memory are then freed up (deallocated) so that they can be reallocated. This process is often referred to as *garbage collection*. The line

```
int vec[] = new int[4];
```

can also be written as

```
int[] vec = new int[4];
```

To access array elements one uses the index notation and the rectangular bracket `[]`. For example

```
int result = vec[1]*vec[2];
```

The class `Arrays` can be used to do operations on arrays of basic data types, for example

```
equals(), fill(), sort(), binarySearch()
```

```
// Marray.java
// one-dimensional arrays

public class Marray
{
  public static void main(String[] args)
  {
  int intnumbers[] = { -2, 4, 5, 6, 7, 11, -19, -23, 89, -9 };
  // the line
  // int intnumbers[10] = { -2, 4, 5, 6, 7, 11, -19, -23, 89, -9 };
  // would give the error message:
  // can't specify array dimension in a type expression

  int sum = 0;
  for(int i=0;i<intnumbers.length;i++)
  {
  sum += intnumbers[i];
  }
  System.out.println(sum);

  double doublenumbers[] = new double[5];
  doublenumbers[0] = 3.1;
  doublenumbers[1] = -1.7;
  doublenumbers[2] = 4.5;
  doublenumbers[3] = 6.7;
  doublenumbers[4] = 7.8;
  double add = 0.0;
  for(int j=0;j<doublenumbers.length;j++)
  {
  add += doublenumbers[j];
  }
  System.out.println(add);
  }
}
// sum += intnumbers[i];
// is the short cut for
// sum = sum + intnumbers[i];
```

```
// Histogram.java
// Histogram printing program

import java.awt.Graphics;
import java.applet.Applet;

public class Histogram extends Applet
{
  int n[] = { 19, 3, 15, 7, 11, 9, 13, 5, 17, 1 };

  // paint the Applet
  public void paint(Graphics g)
  {
  int xPosition;
  int yPosition = 25;

  g.drawString("Element",24,yPosition);
  g.drawString("Value",100,yPosition);
  g.drawString("Histogram",175,yPosition);

  for(int i=0;i<n.length;i++)
  {
  yPosition += 15;
  g.drawString(String.valueOf(i),25,yPosition);
  g.drawString(String.valueOf(n[i]),100,yPosition);

  xPosition = 175;

  for(int j=1;j<=n[i];j++)
  {
  g.drawString("*",xPosition,yPosition);
  xPosition += 7;
  } // end for loop j
  } // end for loop i
  } // end paint
}
```

```
// Matrix.java
// two-dimensional arrays

public class Matrix
{
   public static void main(String[] args)
   {
   int mat[][] = new int[2][2];
   mat[0][0] = -1;
   mat[0][1] = 1;
   mat[1][0] = 2;
   mat[1][1] = -7;

   int trace = 0;
   trace = mat[0][0] + mat[1][1];
   System.out.println(trace);

   int length1 = mat.length;
   int length2 = mat[0].length;

   System.out.println("length1 = " + length1); // => 2
   System.out.println("length2 = " + length2); // => 2

   int array[][] = { { -1, 1 }, { 2, -7 }};
   int result = 0;
   for(int i=0;i<2;i++)
   {
   for(int j=0;j<2;j++)
   {
   result += array[i][j];
   }
   }
   System.out.println(result);
   }
}
// in C++ we must apply twice the
// new operator to allocate the memory.
//
// The line
// int array[2][2] = {{ -1, 1 }, { 2, -7 }};
// is not allowed.
// Error message. Can't specify array dimension
// in type expression.
```

```
// ThreeArr.java
// three dimensional arrays

public class ThreeArr
{
   public static void main(String[] args)
   {
   int three[][][] = new int[2][2][2];
   three[0][0][0] = -1;
   three[0][0][1] = 1;
   three[0][1][0] = -9;
   three[0][1][1] = 11;
   three[1][0][0] = 2;
   three[1][0][1] = -7;
   three[1][1][0] = 20;
   three[1][1][1] = 8;

   int sum = 0;
   int i, j, k;
   for(i=0;i<=1;i++)
   for(j=0;j<=1;j++)
   for(k=0;k<=1;k++)
   {
   sum += three[i][j][k];
   }
   System.out.println(sum);

   int array[][][] =
   {{{ -1, 1 }, { -9, 11 }}, {{ 2, -7 }, { 20, 8 }}};
   int result = 0;
   for(i=0;i<=1;i++)
   for(j=0;j<=1;j++)
   for(k=0;k<=1;k++)
   {
   result += array[i][j][k];
   }
   System.out.println(result);
   }
}
// In C++ the new operator
// must be applied three times
// to allocate the memory.
```

Arrays of basic data types in Java are classes. This means the array identifier is actually a handle to a true object that is created on the heap. The heap object can be created either implicitly, as part of the array initialization syntax, or explicitly with a `new` expression.

The following two programs demonstrate this behaviour. In the first program we override the method

```
protected Object clone()
```

from the `Object` class to make a copy of the array. Overriding occurs when a method is defined in a class and also in one of its subclasses.

Class `Object` is the root of the class hierarchy. Every class has `Object` as a superclass. All objects, including arrays, implement the methods of this class.

In the second program we show how assignment of two arrays works. Assignment of two arrays can be very dangerous as the output of the program shows.

```
// ArrClone.java

public class ArrClone
{
  public static void main(String[] args)
  {
  int[] vec = new int[3];
  vec[0] = 2; vec[1] = 4; vec[2] = -5;

  int[] vecClone = (int[]) vec.clone();

  for(int i=0;i<vecClone.length;i++)
   System.out.println(vecClone[i]);
  }
}
```

```
// ArrAssign.java

public class ArrAssign
{
   public static void main(String args[])
   {

   double x[] = { 2.2, 4.5, 3.1 };
   double y[] = { 4.1, 5.6, -8.9 };

   y = x;

   int i;
   for(i=0;i<y.length;i++)
   {
   System.out.println("y[" + i + "] = " + y[i]);
   }

   x[0] = -17.4;

   for(i=0;i<y.length;i++)
   {
   System.out.println("y[" + i + "] = " + y[i]);
   }
   }
}
// output
// y[0] = 2.2    y[1] = 4.5  y[2] = 3.1
// y[0] = -17.4  y[1] = 4.5  y[3] = 3.1
```

2.8 Control Statements

2.8.1 Introduction

Control statements control the program flow. For example, selection statements such as `if ... else` and `switch` use certain criteria to select a course of action within a program. Iterative control statements (like `for`, `while`, `do ... while`) on the other hand see to it that under certain conditions control is passed from the last statement in a block to the first statement. The use of these control statements is illustrated in the following sections. To apply the control statements we first have to introduce the *relational operators*. Relational operators allow the user to compare two values yielding a result based on whether the comparison is true or false. If the comparison is false then the resulting value in Java is `false` and in C, C++ is 0; if true the value is `true` in Java and in C, C++ 1. The relational operators in Java, C and C++ for the basic data types `byte`, `short`, `int`, `long`, `float`, `double`, `char` are

```
>   greater than
>=  greater than or equal to
<   less than
<=  less than or equal to
==  equal to
!=  not equal to
```

For the basic data type `boolean` we have

```
==  equal to
!=  not equal to
```

For example

```
int x = 234567;
int y = 234568;
if(x == y) ...

boolean b1 = false;
boolean b2 = true;
if(b1 != b2) ...
```

To compare strings the operator `==` should not be used instead we have to use the methods `equals()`.

Selection or flow-control statements select from alternative courses of action by testing certain values. There are two types of selection statements: the `if ... else` and the `switch`.

2.8.2 The if Statement

The `if` statement consists of the `if` keyword followed by a test expression in parentheses, and a single or compound statement. The statement is executed if the test expression is `true`, or skipped if the expression is `false`. The `if` statement has the following pattern

```
if(cond-expression) t-st <else f-st>
```

The `cond-expression` must be of scalar type. The expression is evaluated. If the value is zero (or `NULL` for pointer types in C and C++), we say that the `cond-expression` is false; otherwise, it is true. If there is no `else` clause and `cond-expression` is true, `t-st` is executed; otherwise, `t-st` is ignored. If the optional `else f-st` is present and `cond-expression` is true, `t-st` is executed; otherwise, `t-st` is ignored and `f-st` is executed. Java has a boolean data type. C does not have a specific Boolean data type and only newer C++ compiler have a data type `bool`. Any expression of integer or pointer type can serve a Boolean role in conditional tests. In Java the expression is always evaluated to `true` or `false`. The relational expression `(a > b)` (if legal) evaluates to `int` 1 (true) if `(a > b)`, and to `int` 0 (false) if `(a <= b)`.

```
// Mif.java

class Mif
{
     public static void main(String[] args)
     {
     int x = -5;
     if(x != 0)
     System.out.println("value is nonzero");  // => value is nonzero

     int y = 7;
     if(y > 0)
     System.out.println("positive");
     else System.out.println("negative");    // => positive

     char c1 = 'a';  char c2 = 'b';
     if(c1 == c2)
     System.out.println("the same");
     else System.out.println("not the same"); // => not the same

     double u = 3.140;   double v = 3.141;
     boolean b = (u != v);
     System.out.println("boolean value " + b); // => true
     }
}
```

2.8.3 The for Loop, while Loop, do-while Loop

Iteration statements let us loop a set of statements. There are three forms of iteration in Java, C and C++: `while`, `do-while`, and `for` loops.

The general format for the `while` statement is

```
while(cond-exp) t-st
```

The loop statement, t-st, will be executed repeatedly until the conditional expression, cond-exp, compares to zero (false). The cond-exp is evaluated and tested first. If this value is nonzero (true), t-st, is executed. If no jump statements that exit from the loop is encountered, cond-exp is evaluated again. This cycle repeats until cond-exp is zero.

The `while` loop offers a concise method for scanning strings (end of string indicate by the null character `'\0'`) and other null-terminated data structures.

The general format for the `do-while` statement is

```
do do-st while(cond-exp);
```

The `do-st` statement is executed repeatedly until `cond-exp` compares equal to zero (false). The main difference from the `while` statement is that `cond-exp` is tested after, rather than before, each execution of the loop statement. At least one execution of `do-st` is assured.

The `for` statement format in Java, C and C++ is

```
for(<init-exp>;<test-exp>;<increment-exp>) statement
```

The sequence of events is as follows. The initializing expression `init-exp`, if any, is executed. This usually initializes one or more loop counters, but the syntax allows an expression of any degree of complexity (including declarations in Java, C and C++). The expression `test-exp` is evaluated following the rules of the `while` loop. If `test-exp` is non-zero (true), the loop statement is executed. An empty expression here is taken as `while(1)`, that is, always true. If the value of `test-exp` is zero (false), the `for` loop terminates. The statement `increment-exp` advances one or more counters. The expression `statement` (possible empty) is evaluated and control returns to the expression `test-exp`.

One can break out of a block of statements (typically an iterative block) using Java's `break` statement. This is usually not the preferred method and should only be used if other methods are excessively cumbersome.

The `while` loop is the most general loop and can be used to replace the other two; in other words, a `while` loop is all we need, and the others are just there for our convenience.

```
// Loops.java
// for loop, while loop, do-while loop
// we apply the for-loop, while-loop and do-while-loop
// to find the sum 1 + 2 + 3 + ... + 9 + 10

class Loops
{
   public static void main(String[] args)
   {
   int n = 10;

   int s1 = 0;
   int j;
   for(j=1;j<=n;j++)
   {
   s1 = s1 + j;   // can also be written as s1 += j;
   }
   System.out.println(s1);  //=>  55

   int i = 0;
   int s2 = 0;
   while(i <= n)
   {
   s2 = s2 + i;
   i++;
   }
   System.out.println(s2);  // => 55

   int k = 0;
   int s3 = 0;
   do
   {
   s3 += k;
   k++;
   } while(k <= n);
   System.out.println(s3); // => 55
   }
}
```

In the following program we use three `for` loops to generate all combinations for three capital letters, i.e.

```
AAA, AAB, ... , ZZY, ZZZ
```

There are

$$26 \times 26 \times 26 = 17576$$

combinations. We compare them to a given string consisting of three capital letters. If the string is found we display it. The method `equals()` in the `String` class is used.

```
// Password.java

public class Password
{
   public static void main(String[] args)
   {
   String password = new String("XYA");
   int i, j, k;
   for(i=65;i<91;i++)
   {
   for(j=65;j<91;j++)
   {
   for(k=65;k<91;k++)
   {
   char c1 = (char) i;  // type conversion ASCII table
   char c2 = (char) j;
   char c3 = (char) k;
   char data[] = { c1, c2, c3 };   // array of characters
   String s = new String(data);    // conversion to String
   boolean found = password.equals(s);
   if(found == true)
   {
   System.out.println("password = " + s);
   System.exit(0);
   } // end if
   } // end k-for loop
   } // end j-for loop
   } // end i-for loop
   } // end main
}
```

A *polygon* is a closed plane figure with n sides. If all sides and angles are equivalent the polygon is called regular. The area of a planar convex polygon with vertices

$$(x_0, y_0),\ (x_1, y_1),\ \ldots, (x_{n-1}, y_{n-1})$$

is given by

$$A = \frac{1}{2}\sum_{i=0}^{n-1}(x_i y_{i+1} - x_{i+1}y_i), \qquad x_n \equiv x_0,\ y_n \equiv y_0$$

The following program finds the area of a given planar convex polygon. The polygon in the program is the unit square. We apply the *modulus operator* `%` to identify n and 0.

```
// Polygon.java

public class Polygon
{
   public static void main(String[] args)
   {
   double[] x = new double[4];
   double[] y = new double[4];

   x[0] = 0.0;  x[1] = 1.0;  x[2] = 1.0;  x[3] = 0.0;
   y[0] = 0.0;  y[1] = 0.0;  y[2] = 1.0;  y[3] = 1.0;

   double area = 0.0;
   int length = x.length;

   for(int i=0;i<x.length;i++)
   {
   area += (x[i]*y[(i+1)%length] - x[(i+1)%length]*y[i]);
   }
   area = 0.5*area;

   System.out.println("area of Polygon = " + area); // => 1.0
   }
}
```

2.8.4 The switch Statement

Often an algorithm contains a series of decisions in which a variable or expression is tested separately for each of the constant integral values it may assume, and different actions are taken. Java, C and C++ provide the switch multiple-selection structure to handle such decision making. The `switch` structure consists of a series of `case` labels, and an optional `default` case.

Thus if one has a large decision tree and all the decisions depend on the value of the same variable we use a `switch` statement instead of a series of `if ... else` constructions. The `switch` statement transfers control to one of several case-labeled statements, depending on the value of the switch expression. Note that if the `break` is omitted, execution will continue over the remaining statements in the switch block. The `break` statement can also be used to break out of an iteration loop.

The `switch` statement uses the following basic format

```
switch(sw-expression) case-st
```

A `switch` statement allows us to transfer control to one of several case-labeled statements, depending on the value of the `sw-expression`. The latter must be of integral type (`byte`, `short`, `int`, `long`) or `char`. In Java, C++ and C it also can be of class type provided that there is an unambiguous conversion to integral type available. Any statement in `case-st` (including empty statements) can be labeled with one or more case labels.

```
case const-exp-i:case-st-i
```

There can also be at most one `default` label.

```
default:default-st
```

After evaluating `sw-expression`, a match is sought with one of the `const-exp-i`. If a match is found, control passes to the statement `case-st-i` with the matching case label. If no match is found and there is a `default` label, control passes to `default-st`. If no match is found and there is no `default` label, none of the statements in `case-st` is executed. Program execution is not affected when `case` and `default` labels are encountered. Control simply passes through the labels to the following statement or `switch`. To stop execution at the end of a group of statements for a particular case, use `break`.

The program illustrates the syntax of the Java selection statement `switch`.

```
// SwitchT.java

import java.awt.*;
import java.io.DataInputStream;
import java.io.IOException;

class SwitchT
{
   public static void main(String[] args) throws IOException
   {
   System.out.println("enter character: ");

   // reading a charater from the keyboard
   DataInputStream s = new DataInputStream(System.in);
   char c = (char) s.readByte();  // type conversion

   // The command
   // char c = s.readChar();
   // will not properly work.
   // It always gives the output
   // you did not enter 'a' or 'b'. Explain why !

   System.out.println(c);

   switch(c)
   {
   case 'a':
      System.out.println("you entered 'a' ");
      break;

   case 'b':
      System.out.println("you entered 'b' ");
      break;

   default:
      System.out.println("you did not enter 'a' or 'b' ");
      break;
   }
   }
}
```

The method `char charAt(int)` is in the `String` class and returns the character at the position specified in the argument counting from zero. The method `int length()` is in the `String` class and finds the length of a string.

```
// MSwitch.java

public class MSwitch
{
   public static void main(String[] args)
   {
   String s = new String("bella");

   for(int i=0;i<s.length();i++)
   {
   char c = s.charAt(i);
   System.out.print("i = " + i);

   switch(c)
   {
   case 'a': System.out.println("character is 'a' ");
      break;
   case 'b': System.out.println("character is 'b' ");
      break;
   default: System.out.println("character is not 'a' or 'b' ");
      break;
   }
   }

   int array[] = { 3, 4, 1, 1, 4, -3, 1, 6 };

   for(int j=0;j<array.length;j++)
   {
   switch(array[j])
   {
   case 1: System.out.println("number is 1");
     break;
   case 3: System.out.println("number is 3");
     break;
   default: System.out.println("number is not 1 or 3");
     break;
   }
   }
   }
}
```

2.9 Logical AND, Logical OR and Logical NOT

The *logical operators* in Java, C and C++ are

```
&&  logical AND
||  logical OR
!   logical NOT
```

The logical operators work with logical values (`true` and `false`) allowing us to combine relational expressions. In Java the logical operators always produce a result of either `false` or `true`. In C and C++ we have 0 and 1. The logical operators `&&` and `||` are short circuit. Suppose that we have the expression

```
exp1 && exp2
```

If `exp1` is `false`, then the entire expression is `false`, so `exp2` will never be evaluated. Likewise given the expression

```
exp1 || exp2
```

`exp2` will never be evaluated if `exp1` is true. As an example consider

```
int x = -5;
int y = 7;
boolean result = ((x > 0) && (y > 0));
System.out.println(result); // => false
```

The first expression is `false` thus the second expression is not evaluated.

Warning. Look out for the operator precedence. For example the relational operators (`< > ==`) have higher precedence than the logical AND operator. Note that

```
if(!n)
```

is equivalent to

```
if(n==0)
```

The logical operators should not be confused with the bitwise operators `&`, `|`, `~`. In Java, C and C++ the bitwise AND operator is `&`, the bitwise OR operator is `|`, and the bitwise NOT is `~`.

```
// Logic.java

public class Logic
{
   public static void main(String[] args)
   {
   int a = 5;
   if((a > 0) && (a%2 == 0))
   System.out.println("number is positive and even");

   int b = 7;
   if((b > 0) || (b%2 == 0))
   System.out.println("number is positive or even");

   char d = 'A';
   char e = 'B';
   char f = 'C';
   if((d != e) && (d != f) && (e != f))
   System.out.println("characters are different");

   int c = 1;
   if(!(c == 0))
   System.out.println("number is nonzero");

   int x = (int) (Math.random()*100.0);
   int y = (int) (Math.random()*100.0);
   boolean bool = ((x > 10) && (y < 50));
   System.out.println("bool = " + bool);
   }
}
```

2.10 Bitwise Operations

The *bitwise operations* in Java, C and C++ are: bitwise AND `&`, bitwise OR `|`, bitwise XOR `^` and bitwise NOT `~`. To understand the bitwise operation we recall that every integer number can be written in *binary notation*. For example the integer number 14 (i. e. base 10) can be written as

$$14 = 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 \, .$$

Thus in binary we can express 14 (base 10) as bitstring 1110. If 14 is of type `int` (4 bytes = 32 bits) the binary representation is `00000000000000000000000000001110`. The bitwise operations are as follows.

```
bitwise AND &
1 1 => 1
0 1 => 0
1 0 => 0
0 0 => 0

bitwise inclusive OR |
1 1 => 1
1 0 => 1
0 1 => 1
0 0 => 0

bitwise exclusive OR ^ (XOR)
1 1 => 0
1 0 => 1
0 1 => 1
0 0 => 0

bitwise NOT operator ~ (one's complement)
0 => 1
1 => 0
```

As an example consider the two (decimal) numbers 14 and 9 of data type `int`, i.e. the size is 4 bytes (32 bits). In binary 9 is given by 1001. Thus the bitwise AND of these integers is

```
00000000 00000000 00000000 00001001
00000000 00000000 00000000 00001110
-----------------------------------
00000000 00000000 00000000 00001000
```

Thus the result of the bitwise AND operation is $8 (= 2^3)$ in decimal.

All modern CPU's use the *two complement* to find the negative number of a given integer number, for example `23 -> -23` and `-14 -> 14`. The two complement consists of the one complement ~ and then adding a 1 to the least significant bit.

```
// Bitwise.java

public class Bitwise
{
   public static void main(String[] args)
   {
   int r1 = 14;  // binary 1110
   int r2 = 7;   // binary  111
   int r3 = r1 & r2;
   System.out.println(r3); // => 6 = 110binary
   int r4 = r1 | r2;
   System.out.println(r4); // => 15 = 1111binary
   int r5 = r1 ^ r2;
   System.out.println(r5); // => 9 = 1001binary

   // using XOR ^ to initialize an integer to zero
   int r6;
   r6 = 234;
   r6 = r6 ^ r6;
   System.out.println(r6); // => 0

   // bitwise NOT (one complement)
   int r7 = ~r1;
   System.out.println(r7); // => -15

   // capital letter to small letter conversion using OR
   char c = 'A';
   int ic = (int) c;
   ic = ic | 32;
   c = (char) ic;
   System.out.println("c = " + c);  // => c = a

   // two complement to find the negative number
   // of a given number
   // The operation ~ gives the one complement
   // and then we add 1 to find the two complement
   int r8 = (~r1) + 1;
   System.out.println(r8); // => -14
   }
}
```

2.11 Shift Operations

The *shift operators* in Java, C and C++ are << and >>. The left shift operator is << and the right shift operator is >>. In the expression `E1 << E2` and `E1 >> E2`, the operands `E1` and `E2` must be of integral type. The normal integral type conversions are performed on `E1` and `E2`, and the type of the result is the type of the type converted `E1`. If `E2` is negative or is greater than or equal to the width in bits of `E1`, the operation is undefined. The result of `E1 << E2` is the value `E1` left-shifted by `E2` bit positions, zero-filled from the right if necessary. If `E1` is of signed type, the fill from the left uses the sign bit (0 for positive, 1 for negative `E1`). This sign-bit extension ensures that the sign of `E1 >> E2` is the same as the sign of `E1`. Except for signed types, the value of `E1 >> E2` is the integral part (integer division) of the quotient $E1/2^{E2}$. Java adds the triple right shift >>> to act as a logical right shift by inserting zeroes at the top end; the >> inserts the sign bit as it shifts (an arithmetic shift).

```
// Shift.java

public class Shift
{
   public static void main(String[] args)
   {
   int r1 = 7; // binary 111
   int r2 = (r1 << 1);
   System.out.println(r2);  // binary 1110 = decimal 14
   // shift by 1 is nothing else then
   // multiplication by 2

   int r3 = 3;   // binary 11
   int r4 = (r3 << 3);
   System.out.println(r4);  // => binary 11000 = decimal 24 = 3*8
   // shift by 3 is multiplication by 8 = 2^3

   int r5 = 18;  // binary 10010
   int r6 = (r5 >> 2);
   System.out.println(r6);  // => binary 100 = decimal 4
   // integer division by 4 = 2^2

   int r7 = (16 >> 5);  // decimal 16 = 10000 binary
   System.out.println(r7);  // => 0 decimal
   }
}
```

2.12 Pass by Value, Pass by Reference

In Java basic data types such as `double` can only be passed by value to methods. Java has no pointers and references. In C and C++ arguments to functions can be passed either by value or by reference. We can use pointers or references in C++ to pass values by reference. When an argument is passed by value, a copy of the argument is produced, and the associated parameter is the same as a local variable for the function. This means, changes to the parameter variable will have no effect on the original argument value. Functions that receive variables as parameters get local copies of those variables, not the originals. The alternative, pass by reference in C++, is indicated by the presence of the ampersand operator `&` in the argument list. When arguments are passed by reference, the parameter variable is an alias for the argument value. Thus, changes in the parameter also alter the original argument. In C and obviously in C++ we also can use pointers and the dereference operator to pass by reference. Thus functions that receive pointers to variables gain access to the original variables associated with the pointers. Java, C and C++ pass all arrays by reference.

The following program we pass two integers to the function `swap`. Obviously no swapping of the two numbers takes place since we pass by value.

```
// Refer.java

class Refer
{
   public static void swap(int x,int y)
   {
   int temp;
   temp = x; x = y; y = temp;
   }

   public static void main(String[] args)
   {
   int x = 7;
   int y = 3;

   swap(x,y);

   System.out.println(x); // => 7
   System.out.println(y); // => 3
   }
}
```

```
// Trick.java
// swapping two integers
// arrays passed by reference

public class Trick
{
   public static void swap(int[] a,int[] b)
   {
   int temp;
   temp = a[0];
   a[0] = b[0];
   b[0] = temp;
   }

   public static void main(String[] args)
   {
   int[] x = new int[1];  x[0] = 7;
   int[] y = new int[1];  y[0] = 3;
   System.out.println("before swapping");
   System.out.println("x[0] = " + x[0]);
   System.out.println("y[0] = " + y[0]);

   swap(x,y);

   System.out.println("after swapping");
   System.out.println("x[0] = " + x[0]); // => 3
   System.out.println("y[0] = " + y[0]); // => 7
   }
}
```

The `swap` function can also be written as

```
public static void swap(int[] a,int[] b)
{
  a[0] = a[0] ^ b[0];
  b[0] = a[0] ^ b[0];
  a[0] = a[0] ^ b[0];
}
```

where `^` is the bitwise XOR operation. This avoids the introduction of the temporary (local) variable `temp`.

```
// Shell.java

class Shell
{
   public static void sort(int[] a)  // can the public be omitted ?
   {
   int n = a.length;    // finds length of array
   int incr = n/2;
   while(incr >= 1)
   {
   for(int i=incr;i<n;i++)
   {
   int temp = a[i];
   int j = i;
   while((j >= incr) && (temp < a[j-incr]))
   {
   a[j] = a[j-incr];
   j -= incr;
   }
   a[j] = temp;
   }
   incr /= 2;    // short cut for: incr = incr/2;
   }
   }

   public static void main(String[] args)
   {
   // make a one-dimenional array of ten integers
   int[] a = new int[10];

   int i;
   // fill the array with random numbers
   for(i=0;i<a.length;i++)
   a[i] = (int)(Math.random()*100);

   // call sorting function
   sort(a);

   // print the sorted array
   for(i=0;i<a.length;i++)
   System.out.println(a[i]);
   }
}
// random() is a method in class Math random number between 0 and 1
```

Java passes everything by value. When we are passing basic data types into a method, we get a distinct copy of the basic data type. When we are passing a handle into a method, we get a copy of the handle. In other words Java manipulates objects by reference, but it passes object references to methods by value. The following program demonstrates this.

```
// PassString.java

public class PassString
{
   public static void main(String[] args)
   {
   String str1 = new String("27");
   String str2 = new String("11");

   String[] str3 = new String[1];
   str3[0] = "45";

   String[] str4 = new String[1];
   str4[0] = "88";

   surprise(str1,str2,str3,str4);

   System.out.println("str1 = " + str1);      // => 27

   System.out.println("str2 = " + str2);  // => 11

   System.out.println("str3[0] = " + str3[0]);  // => 44

   System.out.println("str4[0] = " + str4[0]); // => 87
   }

   static void surprise(String str1,String str2,String[] str3,
                        String[] str4)
   {
   str1 = "26";
   str2 = new String("10");
   str3[0] = "44";
   str4[0] = new String("87");
   }
}
```

In the following program the calling method and the called method have an object in common, and both methods can change the object. The object reference `arg1` is passed by value. Then a copy of it is made into the stack frame for the method `inverse()`. But both the original and the copy are object references, and they point to a common object in memory that can be modified.

```
// Reference.java

class Help
{
   public double y;

   public Help(double y)
   {
   this.y = y;
   }

   public String toString()
   {
   return Double.toString(y);
   }
}

class Reference
{
   static void inverse(Help arg1)
   {
   arg1.y = -arg1.y;
   }

   public static void main(String[] args)
   {
   Help arg1 = new Help(3.14159);
   inverse(arg1);
   System.out.println("arg1 = " + arg1); // => -3.14159
   }
}
```

2.13 Recursion

Recursion plays a central role in computer science. A recursive function is one whose definition includes a call to itself. More generally, a recursive method is a method that calls itself either directly or indirectly through another method. A recursion needs a stopping condition. Of course, it is not allowed to use the `main` function in a recursive call. Java, C and C++ allow recursion.

We consider three examples. In the first example we show how the factorial function can be implemented using recursion. The second example shows an implementation the recursively display of elements of an array. In the third example the Fibonacci numbers are calculated using recursion.

```
// Recur.java

class Recur
{
   public static long fac(long n)
   {
   if(n > 1L)
   return n*fac(n-1L);
   else
   return 1L;
   }

   public static void main(String[] args)
   {
   long n = 12L;
   long result = fac(n);
   System.out.println(result);  // => 479001600
   }
}
```

```
// Recursively display the elements of an array

import java.awt.*;
import java.applet.Applet;

public class RDisplay extends Applet
{
   int yPosition = 0;
   int a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

   public void paint(Graphics g )
   {
   someFunction(a,0,g);
   }

   public void someFunction(int b[],int x,Graphics g)
   {
      if(x < b.length)
      {
      someFunction(b,x+1,g);
      g.drawString(String.valueOf(b[x]),25,yPosition);
      yPosition += 15;
      }
   }
}
```

The HTML file `RDisplay.html` is as follows:

```
<HTML>
<APPLET code="RDisplay.class" width=275 height=150>
</APPLET>
</HTML>
```

The *Fibonacci numbers* are defined by the recurrence relation

$$x_{t+2} = x_{t+1} + x_t$$

where $t = 0, 1, 2, \ldots$ and $x_0 = 0$, $x_1 = 1$. We find

$$x_2 = 1, \quad x_3 = 2, \quad x_4 = 3, \quad x_5 = 5, \quad x_6 = 8, \ldots$$

Thus we can apply recursion to obtain the Fibonacci numbers. It is left as an exercise to rewrite the program using iteration.

```
// Fibo.java

public class Fibo
{
   public static long fib(long n)
   {
   if(n == 0) return 0;
   if(n == 1) return 1;

   return fib(n-1) + fib(n-2);
   }

   public static void main(String[] args)
   {
   long n = 10;
   long result = fib(n);
   System.out.println("result = " + result);
   }
}
```

2.14 Jump Statements

A jump statement, when executed, transfers control unconditionally. There are four such statements

```
break, continue, goto, return
```

However `goto` cannot be used in Java. A `break` statement can be used only inside an iteration (`while`, `do-while` and `for` loops) or `switch` statement. It terminates the iteration or `switch` statement. Since iteration and switch statements can be intermixed and nested to any depth, we have to take care that the `break` exits from the correct loop or switch. The rule is that a `break` terminates the nearest enclosing iteration or `switch` statement.

The syntax for the `continue` statement is

```
continue
```

A `continue` statement can be used only inside an iteration statement. It transfers control to the test condition for `while` and `do while` loops, and to the increment expression in a `for` loop. With nested iteration loops, a `continue` statement is taken as belonging to the nearest enclosing iteration.

Java lacks a `goto` statement, although there is a keyword `goto`. In C and C++ the syntax for the `goto` statement is

```
goto Label
```

The `goto` statement transfers control to the statement labeled `Label`, which must be in the same function. One good use for the `goto` is to exit from a deeply nested routine. A simple `break` statement would not work here, because it would only cause the program to exit from the innermost loop. In Java nested `do-while` loops can replace the `goto`'s. The following programs one in C++ using `goto`'s and one in Java using `do-while` demonstrate this.

Unless the function return type is `void`, a function body must contain at least one return statement with the following format

```
return return-expression;
```

where return-expression must be of type `type` or of a type that is convertible to `type` by assignment. The value of the return-expression is the value returned by the function.

```
// didact.cpp
// C++ program using goto

#include <iostream.h>
#include <stdlib.h>
#include <time.h>

int main()
{
   time_t t;
   srand((unsigned long) time(&t));  // seed

   L3:
   int a = rand()%100;
   int b = rand()%100;
   int result;
   L1:
   cout << a << " + " << b << " = ";
   cin >> result;
   if(result == (a+b))
   goto L2;
   cout << "sorry you are not correct: try again" << endl;
   goto L1;
   L2:
   cout << "congratulations you are correct" << endl;
   char c;
   cout << "Do you want to add again: Press y for yes and n for no: ";
   cin >> c;

   if(c == 'y') goto L3;
   if(c == 'n')
   {
   cout << "bye, see you next time around";
   goto L4;
   }
   L4:

   return 0;
}
```

The C++ program can be rewritten in Java as follows:

```
// Gotorau.java
//
// public String readLine() throws IOException
// reads a line of text
//
// int Integer.parseInt(String)
// parses the string argument as a signed decimal integer

import java.io.*;

class Gotorau
{
   public static void main(String[] args) throws IOException
   {
   BufferedReader kbd;
   kbd = new BufferedReader(new InputStreamReader(System.in));
   String input;

   do
   {
   int a = randomInt(0,99);  int b = randomInt(0,99);
   int r;
   do
   {
   if(b < 0)
   System.out.print(a + " " + b + " = ");
   else
   System.out.print(a + " + " + b + " = ");
   r = Integer.parseInt(kbd.readLine());
   if(r != (a + b))
   System.out.println("try again!");
   } while(r != (a + b));
   System.out.println("Congratulations: correct.");
   System.out.println("Do you want to add again ? (y/n) ");
   input = kbd.readLine();
   } while(input.equals("y"));
   } // end main

   private static int randomInt(int lowerB,int upperB)
   {
   return (int)(Math.random()*(upperB - lowerB) + lowerB);
   }
}
```

2.15 Reading from Keyboard

We read in a string from the keyboard and convert it into an integer. We apply the method `readLine()`. Then we read in another string and convert it to an integer. The constructor

```
BufferedReader(Reader in)
```

creates a buffering character-input stream that uses a default-size input buffer. The method

```
String readLine()
```

in class `BufferedReader` reads a line of text. What happens if we enter: 4.5 ?

```
// Readin1.java

import java.io.*;

class Readin1
{
   public static void main(String[] args) throws IOException
   {
     System.out.print("Enter the first integer = ");
     BufferedReader kbd =
     new BufferedReader(new InputStreamReader(System.in));
     String str1 = kbd.readLine();

     // convert String to int
     Integer A = new Integer(str1);
     int a = A.intValue();

     System.out.print("Enter the second integer = ");
     String str2 = kbd.readLine();

     Integer B = new Integer(str2);
     int b = B.intValue();

     int sum = a + b;

     System.out.println("The sum is = " + sum);
   }
}
```

In the following example we read in a string from the keyboard and then convert it to a `double`.

```
// Readin2.java

// reads double (must be positive or zero)
// from keyboard and calculates the square root
// using a do-while loop

import java.io.*;

class Readin2
{
   public static void main(String[] args) throws IOException
   {
     System.out.print("Calculating Square root of a = ");
     BufferedReader kbd =
     new BufferedReader(new InputStreamReader(System.in));
     String str = kbd.readLine();

     // convert String to double
     Double A = new Double(str);
     double a = A.doubleValue();

     double xnew = a/2.0;
     double xold;

     do
     {
     xold = xnew;
     xnew = (xold + a/xold)/2.0;
     System.out.println(xnew);
     }
     while(Math.abs(xnew-xold) > 1E-4);
   }
}
```

The method `abs` is a method in class `Math`. It finds the absolute value of a `double`. What happens if we enter a negative number?

2.16 Command Line Arguments

Java enables us to access command-line arguments by supplying and using the following parameters in method `main`:

```
public static void main(String argv[])
```

Using the command line we compile the following program via

```
javac MCommand.java
```

We then run it with the command

```
java MCommand Good Morning Egoli
```

where

```
argv[0] -> Good
argv[1] -> Morning
argv[2] -> Egoli

// MCommand.java

public class MCommand
{
   public static void main(String argv[])
   {
   System.out.println(argv[2]); // => Egoli
   System.out.println(argv[0]); // => Good

   String t1 = argv[0];
   String t2 = argv[1];
   String t3 = argv[2];

   String t = t1.concat(t2.concat(t3));
   System.out.println("t = " + t); // => GoodMorningEgoli

   String s1 = new String(argv[0]);
   String s2 = new String(argv[1]);
   String s3 = new String(argv[2]);

   String s = s1.concat(s2.concat(s3));
   System.out.println("s = " + s); // => GoodMorningEgoli
   }
}
```

2.17 System Class

The `System` class contains several useful fields and methods. It cannot be instantiated. It contains the method

```
void exit(int status)
```

which terminates the currently running Java Virtual Machine and the method

```
long currentTimeMillis()
```

which returns the current time in milliseconds between the current time and midnight, January 1, 1970 UTC.

Furthermore the class provides standard input, standard output, and error output streams. The fields are

```
static PrintStream err

static PrintStream in

static PrintStream out
```

We can also extract system-dependent information using the method

```
String getProperty(String key)
```

A utility method for quickly copying a portion of an array is also provided. The method

```
void arraycopy(Object src,int src_pos,Object dest,
               int dest_pos,int length)
```

copies an array from the specified source array, beginning at the specified position, to the specified position of the destination array.

The method

```
static SecurityManager getSecurityManager()
```

gets the system security interface.

The following programs show an application of some of these methods.

```
// MySystem.java

import java.util.Properties;

public class MySystem
{
   public static void main(String[] args)
   {
   System.out.println(System.getProperty("os.name"));

   System.out.println(System.getProperty("java.class.path"));

   System.out.println(System.getProperty("os.arch"));

   System.out.println(System.getProperty("user.name"));

   System.out.println(System.getProperty("java.version"));

   long time1 = System.currentTimeMillis();
   for(long i=0;i<100000;i++)
   i++;
   long time2 = System.currentTimeMillis();

   long diff = time2 - time1;
   System.out.println("diff = " + diff);

   int[] a = { 7, 6, 5, 4, 3, 2, 1, 0, 1, 2 };
   int[] b = new int[a.length];

   System.arraycopy(a,0,b,1,a.length-1);
   for(int k=0;k<a.length;k++)
   System.out.println(b[k]);  // => 0 7 6 5 4 3 2 1 0 1

   // to determine the complete set of
   // the current system properties
   // we use the following two lines
   Properties p = System.getProperties();
   System.out.println("p = " + p);
   // the output could be redirected into a file using
   // java MySystem > information.txt
   }
}
```

The method `String getProperty(String)` can be used to find the name of the underlying operating system. The path in different operating systems is different. Thus we can use this information to select the operating system in use.

```
// NoOfLines.java

import java.io.*;
import java.util.*;

public class NoOfLines
{
   public static void main(String[] args) throws IOException
   {
   String s = System.getProperty("os.name");
   System.out.println("s = " + s);
   String fstring = "";

   if(s.equals("Windows 98"))
   {
   fstring = "c:\\books/db/data.dat";
   }

   if(s.equals("Linux"))
   {
   fstring = "/root/javacourse/data.dat";
   }

   FileInputStream fin = new FileInputStream(fstring);
   BufferedReader in;
   in = new BufferedReader(new InputStreamReader(fin));

   int count = 0;

   while(in.readLine() != null)
   {
   count++;
   }
   System.out.println("count = " + count);

   } // end main
}
```

The `System` class also includes the method

```
public static SecurityManager getSecurityManager()
```

which gets the systems security interface. If a security manager has already been established for the current application, then that security manager is returned; otherwise `null` is returned. The security manager is an abstract class that allows applications to implement a security policy. It allows an application to determine, before performing a possibly unsafe or sensitive operation, what the operation is and whether the operation is being performed by a class created via a class loader rather than installed locally. Classes loaded via a class loader (especially if they have been downloaded over a network) may be less trustworthy than classes from files installed locally. The application can allow or disallow the operation.

```
// Security.java

public class Security
{
   public static void main(String[] args)
   {
   SecurityManager sec = System.getSecurityManager();
   System.out.println("security = " + sec);
   }
}
```

2.18 Assertions

Assertions are a new feature in Java 1.4. An assertion is a program statement containing a boolean expression, that a programmer believes to be true at the time the statement is executed. Thus the new keyword `assert` is introduced and support in the library `java.lang.AssertionError`. The following program gives an example. We compile with

```
javac -source 1.4 MyAssert.java
```

and run it with

```
java -ea MyAssert
```

The `-ea` switch is used to enable assertions at run time.

```
public class MyAssert
{
   static double scalar(double[] v,double[] w)
   {
   assert v.length == w.length : "not the same length";

   double temp = 0.0;
   for(int i=0;i<v.length;i++)
   temp += v[i]*w[i];
   return temp;
   }

   public static void main(String args[])
   {
   double[] v = new double[2];
   v[0] = 1.2;  v[1] = 2.3;
   double[] w = new double[3];
   w[0] = 4.5;  w[1] = 3.1;  w[2] = 7.1;
   double result = 0.0;
   try
   {
   result = scalar(v,w);
   System.out.println("result = " + result);
   }
   catch(AssertionError e)
   {
   System.out.println("exception: " + e);
   }
   }
}
```

2.19 Applets and HTML Parameters

The following example shows how to use HTML parameters in an Applet using the `getParameter()` method.

```
// Parameters.java

import java.awt.*;
import java.applet.*;

public class Parameters extends Applet
{
   // save the first HTML parameter as a String
   String par1;
   // second one should be an int
   int par2;
   // third one should be an int
   int par3;
   // we add par2 and par3
   int sum;

   public void init()
   {
   // method gets the specified parameter's value
   // out of the HTML code that is calling the applet
   par1 = getParameter("parameter1");
   // since parameter2 and parameter3 are read as string
   // we need to transform them to integers using
   // the wrapper class Integer
   par2 = Integer.parseInt(getParameter("parameter2"));
   par3 = Integer.parseInt(getParameter("parameter3"));
   sum = par2 + par3;
   }

   public void paint(Graphics g)
   {
   // shows what is in the HTML parameter code
   g.drawString("Parameter 1 is: " + par1,20,20);
   g.drawString("Parameter 2 is: " + par2,20,40);
   g.drawString("Parameter 3 is: " + par3,20,60);
   g.drawString("Parameter 2 + Parameter 3 is: " + sum,20,80);
   }
}
```

The HTML file is given by

```
<HTML>
<APPLET CODE="Parameters.class" width=275 height=135>
<PARAM NAME="parameter1" VALUE="Egoli">
<PARAM NAME="parameter2" VALUE="17">
<PARAM NAME="parameter3" VALUE="-5">
</APPLET>
</HTML>
```

Chapter 3

String Manipulations

3.1 String Class

The `String` class

```
public final class String extends Objects
```

is a general class of objects to represent character Strings. Strings are constant, their values cannot be changed after creation. The compiler makes sure that each `String` constant actually results in a `String` object. A string can be created via two methods

```
String s1 = "abcdef";
String s2 = new String("abcdef");
```

In the second method we use the constructor in the `String` class. Since `String` objects are immutable they can be shared. We demonstrate the use of the following methods

```
int length(), String substring(int beginIndex,int endIndex),
```

```
char charAt(int index), String concat(String str),
```

```
String replace(char oldChar,char newChar), String toLowerCase(),
```

```
String toUpperCase(), boolean equals(Object anObject), String trim()
```

The method `length()` finds the length of this string. The method

```
public int hashCode()
```

returns a hashcode for this string. This is a large number of the character values in the string.

```
// Mstring.java

public class Mstring
{
   public static void main(String[] args)
   {
   String mine = "cde";
   System.out.println("abc" + mine); // => abccde

   String st = new String("summer");
   System.out.println("spring and " + st); // => spring and summer

   // some frequently used methods in the String class
   String city = "egoli";
   int n = city.length();
   System.out.println(n);  // => 5

   String sub = city.substring(0,2);
   System.out.println(sub); // => eg

   char c;
   c = city.charAt(2);
   System.out.println(c);  // => o

   String region = "-Gauteng";
   String str = city.concat(region);
   System.out.println(str); // => egoli-Gauteng

   String name = new String("Otto");
   String new_name = name.replace('t','l');
   System.out.println(new_name);  // => Ollo

   char data[] = { '3', '5', 'c' };
   String d = new String(data);
   System.out.println(d);  // => 35c

   String m = new String("Cape");
   String upper = m.toUpperCase();
   System.out.println(upper);  // => CAPE

   int result = m.hashCode();
   System.out.println(result); // => 3530789

   boolean bool = st.equals(d);
   System.out.println(bool);  // => false
```

```
   // array of Strings
   String keys[] = { "Red", "Green", "Blue" };
   System.out.println(keys[1]);  // => Green

   // removing white space from both ends of this string
   String ws = new String(" xxxyyy ");
   ws.trim();
   System.out.println("ws = " + ws);
   int size = ws.length();
   System.out.println(size); // => 8
   }
}
```

Next we discuss how to convert strings to numbers (integer type or floating point type) and numbers to strings. This technique is important when we read in a number from the keyboard as a string. The abstract class `Number` is the superclass of the wrapper classes

`Byte, Short, Integer, Long, Float, Double`

These subclasses of class `Number` must provide methods to convert the represented numeric value to `byte`, `short`, `int`, `long`, `float` and `double`.

Assuming the specified string represents an integer, the method

```
public static Integer valueOf(String s) throws NumberFormatException
```

returns a new `Integer` object initialized to that value. The method throws an exception if the `String` cannot be parsed as an `int`. The radix is assumed to be 10. The method

```
public int intValue()
```

returns the value of this `Integer` as an `int`. It overrides `intValue()` in class `Number`. Analogously we can consider the methods

`byteValue(), shortValue(), longValue(),`

`floatValue(), doubleValue()`

in the classes `Byte`, `Short`, `Long`, `Float` and `Double`, respectively.

The method

```
public static String toString(int i)
```

in class `Integer` returns a new String object representing the specified integer. The radix is assumed to be 10. Analogously we have the method

```
public static String toString(long l)
```

in the class `Long` etc.

All these classes have the data fields `MAX_VALUE` and `MIN_VALUE`. For example, for the class `Double` `MAX_VALUE` is the largest positive finite value of type `double`.

```
// Convert.java

class Convert
{
   public static void main(String[] args)
   {
   String str;
   str = "25";

   int i = Integer.valueOf(str).intValue();
   System.out.println(" i = " + i);          // => 25

   String s = new String("-45678901");
   long l = Long.valueOf(s).longValue();
   System.out.println(" l = " + l);         // => -45678901

   str = "25.6";
   float f = Float.valueOf(str).floatValue();
   System.out.println(f);                   // => 25.6

   double d = Double.valueOf(str).doubleValue();
   System.out.println(d);                   // => 25.6

   String m;
   double j = 1234.7;
   m = Double.toString(j);
   System.out.println(m);  // => 1234.7

   int k = 3412;
   String n = new String("000000000000000");
   n = Integer.toString(k);
   System.out.println(n);  // => 3412

   String name;
   name = "Olli";
   int x = Integer.valueOf(name).intValue();
   System.out.println(x);  // => NumberFormatException
   }
}
```

3.2 StringTokenizer Class

The class `StringTokenizer` allows an application to break a string into tokens. The `StringTokenizer` methods do not distinguish among identifiers, numbers and quoted strings, nor do they recognize and skip comments.

The constructor

```
public StringTokenizer(String str)
```

constructs a string tokenizer for a specified string. The tokenizer uses the default delimiter set, which is

```
 " \t\n\r\f"
```

the space character, the tab character, the newline character, the carrige-return character, and the form-feed character.

The constructor

```
public StringTokenizer(String str,String delim)
```

constructs a string tokenizer for a specified string. The characters in the `delim` arguments are the delimiters for separating tokens. Delimiter characters themselves will not be treated as tokens.

The method

```
public String nextToken()
```

returns the next token from this `StringTokenizer`. The method

```
public String nextToken(String delim)
```

returns the next token in this string tokenizer's string. The method

```
boolean hasMoreTokens()
```

tests if there are more token available from the tokenizer's string. The method

```
int countTokens()
```

calculates the number of times that this tokenizer's `nextToken()` method can be called before it generates an exception.

The next program shows how the `StringTokenizer` class can be applied.

```
// Tokenizer.java

import java.util.*;

public class Tokenizer
{
   public static void main(String[] args)
   {
   String st = new String("Egoli Gauteng Africa");

   StringTokenizer tok = new StringTokenizer(st);

   String s = tok.nextToken();
   System.out.println(s);        // => Egoli

   s = tok.nextToken();
   System.out.println(s);        // => Gauteng

   s = tok.nextToken();
   System.out.println(s);        // => Africa

   String u = new String("Olli Willi Susi Otto Karl Ludwig Nela");
   String v;
   String name = new String("Nela");
   boolean found = false;
   tok = new StringTokenizer(u);

   int i;
   for(i=1; i <= 7; i++)
   {
   v = tok.nextToken();
   if(v.equals(name))
   {
   found = true;
   break;
   }
   }  // end for loop

   if(found == true)
   System.out.println("name in string");  // => name in string
   else
   System.out.println("name not in string");
```

```
   String comma = new String("123,4567,50001,444");
   String number = new String("50002");

   StringTokenizer tokdel = new StringTokenizer(comma,",");
   String temp;
   boolean in = false;

   int count = tokdel.countTokens();
   System.out.println("count = " + count);

   for(i=1; i <= count; i++)
   {
   temp = tokdel.nextToken();
   if(temp.equals(number))
   {
   in = true;
   break;
   }
   } // end for loop

   if(in == true)
   System.out.println("number in string");
   else
   System.out.println("number not in string");

   } // end main
}
```

When we use the `StringTokenizer` class, we specify a set of delimiter characters; instances of `StringTokenizer` then return words delimited by these characters. However, the class has some limitations, in particular when we parse text that represents human language. The class `java.text.BreakIterator` is a class designed to parse human language text into words, lines and sentences.

3.3 StringBuffer Class

The instances (objects) of the `String` class suffer the restriction of being immutable. This means they cannot be changed. Whenever a string has to be modified we have to construct a `new String` object, either explicitly or implicitly. Java provides the separate `StringBuffer` class for string objects that need to be changed. The reason for this dichotomy is that providing the flexibility for changing a string requires substantial overhead (more space and complexity). In situations where we do not need to change the string, the simpler `String` class is preferred.

The constructor

```
public StringBuffer(int length)
```

constructs a string with no characters in it and an initial capacity specified by the `length` argument. The method

```
public StringBuffer append(String)
```

appends the string to the string buffer.

```
// Buffer.java

public class Buffer
{
   public static void main(String[] args)
   {
   StringBuffer buf = new StringBuffer(20);

   buf.append("Good Morning");
   System.out.println("buf = " + buf);
   System.out.println("buf.length() = " + buf.length());     // => 12
   System.out.println("buf.capacity() = " + buf.capacity()); // => 20

   buf.append(" Egoli Gauteng");
   System.out.println("buf = " + buf);
   System.out.println("buf.length() = " + buf.length());     // => 26
   System.out.println("buf.capacity() = " + buf.capacity()); // => 42
   }
}
```

Chapter 4

Classes and Objects

4.1 Introduction

A Java program is a collection of one or more text files that contains Java classes, at least one of which is `public` and contains a method named `main()` that has this form

```
public static void main(String[] args)
{  ... }
```

A Java class is a specific category of objects, similar to a Java basic data type (e.g. `int`, `double`). Thus a class is a Java aggregate that may contain methods and types in addition to data. It can also be a user-defined data type. A service offered by most classes is that of allowing the user to create instances of the class. These services are called constructors and have the name of the class, no return value and any arguments. The default constructor takes no argument. An unfortunate aspect of Java inherited from C++ is that the compiler writes a default constructor if and only if no other constructors has been defined for the class. A Java class specifies the range of the objects of that class can have. Thus a class includes

```
Constructors
Methods
Fields
```

The programmmer can add his own classes to the built-in classes.

The following tables summarize the modifiers

```
private, public, protected, abstract, final, static,

transient, volatile, native, synchronized
```

that can appear in the declaration of classes, fields, local variables, and methods.

Class Modifiers

Modifier	Meaning
public	It is accessible from all other classes
abstract	The class cannot be instantiated
final	No subclasses can be declared

Constructor Modifiers

Modifier	Meaning
public	It is accessible from all classes
protected	It is accessible only from within its own class and its subclasses
private	It is accessible only from within its own class

Field Modifiers

Modifier	Meaning
public	It is accessible from all classes
protected	It is accessible only from within its own class and its subclasses
private	It is accessible only from within its own class
static	Only one value of the field exists for all instances of the class
transient	It is not part of the persistent state of an object
volatile	It may be modified by asynchronous threads
final	It must be initialized and cannot be changed

Local Variable Modifiers

Modifier	Meaning
final	It must be initialized and cannot be changed

Method Modifiers

Modifier	Meaning
public	It is accessible from all classes
protected	It is accessible only from within its own class and its subclasses
private	It is accessible only from within its own class and its subclasses
abstract	It has no body and belongs to an abstract class
final	No subclasses can override it
static	It is bound to the class itself instead of an instance of the class
native	Its body is implemented in another programming language
synchronized	It must be locked before a thread can invoke it

4.2 Wrapper Classes

Every basic data type has a *wrapper class*

```
byte -> Byte
short -> Short
int -> Integer
long -> Long
float -> Float
double -> Double
char -> Character
boolean -> Boolean
```

Elements in these classes are objects. Each type wrapper class enables us to manipulate basic data types as objects. The `Boolean` class includes the fields `static Boolean TRUE`, `static Boolean FALSE`, and `static Class TYPE` (the class object representing the basic data type `boolean`).

Each of the numeric classes

```
Byte, Short, Integer, Long, Float, Double
```

inherits from the class `Number`. Each of the type wrappers is declared `final`, so their methods are implicitly `final` and may not be overridden.

```
// Wrap.java

class Wrap
{
    public static void main(String[] args)
    {
    Integer i = new Integer(5);
    Integer j = new Integer(123);
    int result = i.intValue() + j.intValue();
    Integer k = new Integer(result);
    System.out.println("result = " + result);  // => 128
    System.out.println("k = " + k);            // => 128

    double x = 3.14;
    double y = 2.79;
    Double X = new Double(x);
    Double Y = new Double(y);
    double sum = X.doubleValue() + Y.doubleValue();
    System.out.println("sum = " + sum);  // => 5.93
    }
}
```

The `Character` class wraps a value of the basic data type `char` in an object. An object of type `Character` contains a single field whose type is `char`. This class provides several methods for determing the type of a character and converting uppercase to lowercase and vice versa. First we look at the constructors and methods in the `Character` class. The constructor

```
Character(char)
```

constructs a `Character` object with the specified value. The method

```
public char charValue()
```

returns the value of this `Character` object. The method

```
public static boolean isDigit(char ch)
```

determines if the specified character is a ISO-LATIN-1 digit. The method

```
public static int digit(char ch,int radix)
```

returns the numeric value of the character digit using the specified radix. The method

```
public boolean equals(Object obj)
```

compares this object against the specified object. Thus the class `Character` overrides the `equals()` method from the `Object` class. The method

```
static char toUpperCase(char ch)
```

converts the character argument to uppercase. Analogously the method

```
static char toLowerCase(char ch)
```

maps the given character to its lowercase equivalent. The class also overrides the method

```
String toString()
```

from the class `Object`. The method

```
int getNumericValue(char ch)
```

returns the numeric value of a character digit, using the value specified in an internal table called the Unicode Attribute Table. For example, `\u217c` is the Roman Numeral L, which has the value of 50.

```
// Charact.java

class Charact
{
   public static void main(String[] args)
   {
   Character k = new Character('#');
   System.out.println(k);              // => #

   char c = k.charValue();
   System.out.println(c);              // => #

   char m = '5';
   Character j = new Character(m);
   boolean b1 = j.isDigit(m);
   System.out.println(b1);             // => true

   char n = '9';
   Character z = new Character(n);
   int a = z.digit(n,10);
   System.out.println(a);             // => 9

   boolean b2 = z.equals(j);
   System.out.println(b2);            // => false

   Character y = new Character('$');
   char x = y.toLowerCase('A');
   System.out.println(x);             // => a

   Character c1 = new Character('&');
   String s1 = c1.toString();
   System.out.println(s1);            // => &

   int number = Character.getNumericValue('\u217c');
   System.out.println("number = " + number);
   }
}
```

4.3 Vector Class

The `Vector` class is a container class. The `Vector` class implements a growable array of objects. It cannot contain basic data types. This means it only can hold `Object` handles. The constructor

```
Vector(int intialCapacity)
```

constructs an empty vector with the specified initial capacity. To insert elements into the container class we apply the method

```
void addElement(Object obj)
```

adds the specified component to the end of this vector. The method

```
Object elementAt(int index)
```

returns the component at the specified index. The method

```
void insertElementAt(Object obj,int index)
```

shifts up elements in order to insert an element. The method

```
void removeElementAt(int index)
```

removes an element and shifts down all elements above it. The method

```
boolean isEmpty()
```

tests if this vector has no components. The method

```
boolean contains(Object obj)
```

tests if the specified object is a component in this `Vector`.

```
// Vec.java

import java.util.*;

class Vec
{
   public static void main(String[] args)
   {
   Vector v = new Vector(5);

   Double x = new Double(3.1);
   Character c = new Character('X');
   Double z = new Double(5.3);
```

```
   Integer n = new Integer(-14);
   String st = new String("New Year");

   v.addElement(x);  // position 0
   v.addElement(c);  // position 1
   v.addElement(z);  // position 2
   v.addElement(n);  // position 3
   v.addElement(st); // position 4

   Character a = (Character) v.elementAt(1);
   System.out.println(a);
   String s = (String) v.elementAt(4);
   System.out.println(s);
   Integer m = (Integer) v.elementAt(3);
   System.out.println(m);

   boolean b = v.isEmpty();
   System.out.println(b);   // => false

   Character aa = new Character('S');
   v.insertElementAt(aa,2);

   Character ch = (Character) v.elementAt(2);
   System.out.println(ch);        // => S

   Double d = (Double) v.elementAt(3);
   System.out.println(d);         // => 5.3
   Character f = (Character) v.elementAt(1);
   System.out.println(f);         // => X

   v.removeElementAt(3);
   Integer zz = (Integer) v.elementAt(3);
   System.out.println(zz);           // => -14
   v.removeElementAt(4);
   String k = new String("New York");
   v.addElement(k);

   boolean b1 = v.contains(k);
   System.out.println(b1);     // => true
   boolean b2 = v.contains("New York");
   System.out.println(b2);     // => true
   boolean b3 = v.contains("New Year");
   System.out.println(b3);    // => false
   }
}
```

4.4 Math Class

The `Math` class contains methods for performing basic numeric operations such as elementary exponential, logarithm, square root and trigonometric functions. The `Math` class includes the fields

```
public static final double PI
public static final double E
```

for the real numbers 3.14159... and 2.71828..., respectively. The class includes the overloaded function

```
int abs(int),  long abs(long),  float abs(float),  double abs(double)
```

The `Math` class also contains all the standard mathematical functions such as

```
sin(),  cos(),  tan(),  acos(),  asin(),  atan(),
```

```
exp(),  log(),  sqrt()
```

The data type passed to these functions is `double` and the return data type is also `double`. Furthermore, we have the functions

```
pow(),  ceil(),  floor(),  max(),  min()
```

and

```
double toDegree(double angrad),  double toRadians(double angdeg)
```

The class also includes a method to generate random number (data type `double`) in the interval $[0, 1]$)

```
public static double random()
```

The following program shows an application of the `Math` class.

```
// MathClass.java

class MathClass
{
   public static void main(String[] args)
   {
   double g = Math.PI;
   System.out.println(" Pi = " + g); // => 3.14159...
   double f = Math.cos(g);
   System.out.println(f);            // => -1.0

   double exp = Math.E;
```

```
   System.out.println(" exp = " + exp); // => 2.71828...

   int a = -7;
   int b = Math.abs(a);
   System.out.println(b);  // => 7

   double c = -8.9;
   double d = Math.abs(c);
   System.out.println(d);  // => 8.9

   long i = Math.max(123L,23456789L);
   System.out.println(i);  // => 23456789

   double r1 = Math.random();
   System.out.println(r1);  // => 0.62872...
   double r2 = Math.random();
   System.out.println(r2);  // => 0.33179...

   double j = Math.pow(Math.E,2.0);
   System.out.println(" j = " + j);   // => 7.38905...

   double k = 7.34;
   double l = Math.floor(k);
   double m = Math.ceil(k);
   System.out.println(l);       // => 7.0
   System.out.println(m);       // => 8.0

   double n = 5.1;
   double p = 2.1;
   double q = Math.IEEEremainder(n,p); // => 0.899999999999995
   // returns the remainder of n divided by p
   // as defined by IEEE 754  => 0.9 (rounding errors)
   System.out.println(q);

   double v = 2.0;
   double z = Math.sqrt(v);
   System.out.println(z);   // => 1.41...

   double r = -2.0;
   double s = Math.sqrt(r);
   System.out.println(s);  // => NaN
   }
}
```

4.5 BitSet Class

The `BitSet` class implements a vector of bits that grows as needed. Each component of the bit set has a boolean value. The bits of the `BitSet` class are indexed by nonnegative integers. Individual indexed bits can be examined, set or cleared.

```
// MyBitSet.java

import java.util.*;   // for BitSet

public class MyBitSet
{
    public static void main(String[] args)
    {
    String[] table = { "sno", "sname", "address", "cno", "cname",
                       "instructor", "office" };

    int size = table.length;
    BitSet bits = new BitSet(size);
    System.out.println("bits = " + bits);

    bits.set(2);  bits.set(5);
    System.out.println("bits = " + bits);

    for(int i=0; i < size; i++)
    {
    if(bits.get(i) == true)
    {
    System.out.println(table[i]);
    }
    }

    BitSet[] bitarray = new BitSet[128];
    for(int k=0; k < 128; k++)
    bitarray[k] = new BitSet(7);

    } // end main
}
```

The output is

```
bits = { }
bits = {2, 5}
address
instructor
```

4.6 BigInteger and BigDecimal Class

The class `BigInteger` provides immutable arbitrary-precision integers. The constructor

```
BigInteger(String val)
```

translates the decimal string representation of a `BigInteger` into a `BigInteger`. The class `BigInteger` contains the data fields

```
public static final ZERO
public static final ONE
```

Thus, for example we can write

```
BigInteger b1 = BigInteger.ZERO;
BigInteger b2 = BigInteger.ONE;
BigInteger b3 = b1.add(b2);
```

Methods we can use with this class are

```
abs(),  add(),  compareTo(),  divide(),  equals(),  gcd(),

negate(),  multiply(),  remainder(),  subtract(),  toString()
```

The class `BigDecimal` provides immutable arbitrary precision signed decimal numbers. The constructor

```
BigDecimal(String val)
```

translates the `String` representation of a `BigDecimal` into a `BigDecimal`. Fields in the class `BigDecimal` are

```
static int ROUND_CEILING
static int ROUND_DOWN
static int ROUND_FLOOR
```

Methods in this class we can use are

```
abs(),  add(),  compareTo(),  divide(),  equals(),  multiply(),

negate(),  subtract(),  toString()
```

The following program shows how to use the `BigDecimal` and `BigInteger` class.

```
// MyDecimal.java

import java.math.*;

public class MyDecimal
{
   public static void main(String args[])
   {
   String t1 = new String("234567");
   String t2 = new String("-5678910009");
   String t3;

   BigInteger b1 = new BigInteger(t1);
   BigInteger b2 = new BigInteger(t2);
   BigInteger b3 = b1.multiply(b2);
   System.out.println("b3 = " + b3);
   t3 = b3.toString();
   System.out.println("t3 = " + t3);

   boolean bool = b2.equals(b1);
   System.outprintln("bool = " + bool);

   BigInteger c1 = BigInteger.ZERO;
   BigInteger c2 = BigInteger.ONE;
   BigInteger c3 = c1.add(c2);
   System.out.println(c3);    // => 1

   String s1 = new String("1.234567890123");
   String s2 = new String("-5.4678912345");
   String s3;

   BigDecimal d1 = new BigDecimal(s1);
   BigDecimal d2 = new BigDecimal(s2);
   BigDecimal d3 = d1.add(d2);
   System.out.println("d3 = " + d3);
   s3 = d3.toString();
   System.out.println("s3 = " + s3);

   BigDecimal d4 = d1.multiply(d2);
   System.out.println("d4 = " + d4);
   }
}
```

4.7 Object Class

Java defines one special class, the `Object` class, which is the ancestor of every other class. It declares twelve members: a constructor and eleven methods. Four of them,

```
clone(), equals(), hashCode(), toString()
```

are intended to be overriden. Thereby they provide clean and consistent facilities for duplicating objects and determining when they are the same.

The class `Object` does not define any data members. It defines only the default constructor (taking no arguments) and a number of methods. Since all classes are directly or indirectly derived from `Object` they all inherit these methods.

The `equals()` method in the class `Object` simply checks whether the object itself and the argument are the same object in the sense they share the same memory position. The method is very often overriden for more meaningful comparisons. For example, the `String` class overides `equals()` for string comparison.

The `clone()` method is Java's copy construtor. It allocates memory for a new instance of the class and makes an identical copy of the object itself. As it is the case for the copy constructor in C++, the default behaviour of the `clone()` method is to make a byte-for-byte copy of our object. If our object contains other objects, the class members are actually pointers to these objects and making a byte-by-byte copy would only copy the pointers and would not create copies of the contained objects. For this reason the `clone()` method is declared `protected` and Java forces override the `clone()` method.

The `toString()` method returns a string which represents the value of the object. The method is overriden frequently. When we call the `print()` or `println()` methods to display an object in text mode on the screen, Java calls the `toString()` method automatically to obtain a string representing the state of the object.

The method `finalize()` is Java's version of the destructor in C++. This method has to be overriden by any class that requires specialized processing when the object is destroyed.

The method `getClass()` identifies the class of an object by returning an object of the class `Class`.

The methods `wait()` and `notify` are used for multithreading and will be discussed later.

```
// MyObject.java

import java.math.*;     // for BigInteger, BigDecimal

public class MyObject
{
   public static void main(String args [])
   {
   Object[] a = new Object[5];     // array of 5 objects
   a[0] = new String("Good Morning");
   a[1] = new Integer(4567);
   a[2] = new Character('X');
   a[3] = new Double(3.14);
   a[4] = new BigInteger("234567890");

   for(int i=0; i<a.length; i++)
   {
   System.out.println(a[i]);
   }

   a[3] = new BigDecimal("11.56789034521111112345");

   System.out.println(a[3]);

   boolean b = a[4].equals(a[3]);
   System.out.println("b = " + b);  // => false

   Object[] x = new Object[1];
   x[0] = new Float(2.89);
   boolean r = x.equals(a);
   System.out.println("r = " + r);  // => false
   }
}
```

The following program shows how to override the

`clone(), equals(Object p), toString()`

methods.

```
// MyClone.java
//
// protected Object clone()  // in Object class
// creates a new object of the same class as this object

class Point
{
   private double x, y;

   Point(double a,double b)  { x = a; y = b; }

   public Object clone()  { return new Point(x,y); }

   public boolean equals(Object p)
   {
   if(p instanceof Point)
   return (x == ((Point) p).x && y == ((Point) p).y);
   else return false;
   }

   public String toString()
   {
   return new String("(" + x + ", "  + y + ")");
   }
}   // end class Point

class MyClone
{
   public static void main(String args[])
   {
   Point p = new Point(2.0,3.0);
   System.out.println("p = " + p);
   Point q = (Point) p.clone();
   System.out.println("q = " + q);

   if(q == p)
   System.out.println("q == p");
   else System.out.println("q != p");         // => q != p

   if(q.equals(p))
   System.out.println("q equals p"); // => q equals p
   else System.out.println("q does not equal p");
   }
}
```

4.8 The this Object

When we want to access the current object in its entirety and not a particular instance variable, Java has a convenient shortcut for this the `this` keyword. In a method, the keyword `this` refers to the object on which the method operates.

```
// Pair.java

public class Pair
{
   private int a, b;

   public Pair(int x)
   {
   a = x;
   b = x + 1;
   }

   public Pair increment_pair()
   {
   a++; b++; return this;
   }

   public void print_pair()
   {
   System.out.print(a);
   System.out.println(b);
   }

   public static void main(String[] args)
   {
   Pair P;
   P = new Pair(19);
   P.increment_pair();
   P.print_pair();           // => 2021

   Pair Q;
   Q = new Pair(55);
   Q.increment_pair();
   Q.print_pair();          // => 5657
   }
}
```

Thus the `this` reference is not instance data stored in an object itself. Rather, the `this` reference is passed into the object as an implicit argument in every non-static method call.

```
// ThisTest.java

import java.awt.Graphics;
import java.applet.Applet;

public class ThisTest extends Applet
{
   private int i = 12;

   public void paint(Graphics g)
   {
   g.drawString(this.toString(),25,25);
   }

   public String toString()
   {
   return "i = " + i + "  this.i = " + this.i;
   }
}
```

The `HTML` file is

```
<HTML>
<APPLET CODE="ThisTest.class" width=275 height=35>
</APPLET>
</HTML>
```

The output is

```
i = 12  this.i = 12
```

The following C++ program demonstrates how the `this` pointer works. The reader should compare `this` and the addresses of the objects `x` and `y`.

```
// myclass.cpp

#include <iostream.h>

class X
{
   public:
      X();
     ~X();
};

X::X()
{
   cout << "this = " << this << endl;
}

X::~X()
{
   cout << "this = " << this << endl;
   cout << "destructor called" << endl;
}

int main()
{
   X x;
   cout << "&x = " << &x << endl;
   X y;
   cout << "&y = " << &y << endl;
   return 0;
}
```

The output is

```
this = 0x0063FDF4
&x   = 0x0063FDF4
this = 0x0063FDF0
&y   = 0x0063FDF0
this = 0x0063FDF0
destructor called
this = 0x0063FDF4
destructor called
```

4.9 The Class Class

The class `Class` has methods for querying constructors, methods and attributes (in the sense of data fields) of a class. Methods, in turn can be further queried for their arguments, return values and any exceptions they might throw.

The method

```
static Class forName(String className)
```

returns the `Class` object associated with the class with the given string name. The method

```
Class[] getClasses()
```

returns an array containing `Class` objects representing all the public classes and interfaces that are members of the class represented by this `Class` object. The methods

```
Constructors[] getConstructors()
Method[] getMethods()
Field[] getFields
```

return an array containing Constructor (Method, Field) objects reflecting all the public constructors of the class represented by this `Class` object.

The method

```
boolean isInstance(Object obj)
```

is dynamic equivalent of the Java language `instanceof` operator. The method

```
boolean isPrimitive()
```

determines if the specified `Class` object represents a primitive (basic) data type. The method

```
Class[] getSuperclass()
```

returns the object that represents the superclass of that class if this object represents any class other than the class `Object`.

The following application takes as command line argument the name of a class and lists the public attributes, constructors and methods of the class.

```
// MyClass.java

import java.lang.reflect.*;

public class MyClass
{
   public static void main(String[] args)
   throws ClassNotFoundException
   {
   Class descriptor = Class.forName(args[0]);

   Constructor[] constructors = descriptor.getConstructors();

   if(constructors.length > 0)
   System.out.println("Constructors = ");

   int i;
   for(i=0; i < constructors.length; ++i)
   System.out.println(constructors[i]);

   Method[] methods = descriptor.getMethods();

   if(methods.length > 0)
   System.out.println("\nMethods = ");
   for(i=0; i < methods.length; i++)
   System.out.println(methods[i]);

   Field[] fields = descriptor.getFields();

   if(fields.length > 0)
   System.out.println("\nFields = ");
   for(i=0; i < fields.length; i++)
   System.out.println(fields[i]);
   }
}
```

4.10 The Calendar Class

The class `Calendar` is an abstract class for converting between a `Date` object and a set of integer fields such as

```
YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, MILLISECOND
```

For the `DAY` we have to select from

```
DAY_OF_MONTH, DAY_OF_WEEK, DAY_OF_WEEK_IN_MONTH, DAY_OF_YEAR
```

A `Date` object represents a specific instant in time with millisecond precision. Subclasses of the `Calendar` class interpret a `Date` according to the rules of a specific calendar system. One concrete subclass of `Calendar` is provided, namely `GregorianCalendar`. The method

```
Calendar getInstance()
```

returns a `GregorianCalendar` object whose time fields given above have been initialized with current date and time. The method

```
int get(int field)
```

in the class `Calendar` gets the value for a given time field.

A `Calendar` object can produce all the time field values needed to implement the date-time formatting for a particular language and calendar style. The interpretation of noon and midnight is as follows: 24:00:00 belongs to the following day and is am, noon belongs to pm.

The `Calendar` class provides support for date conversion that were previously implemented by the `Date` class. The support provided by `Calendar` is more comprehensive and international. The `Calendar` class is an abstract class that can be extended to provide conversions for specific calendar systems. The `GregorianCalendar` subclass supports the predominant `Calendar` class used by many countries. It supports the eras B.C. and A.D. by defining them as class constants. It provides seven constructors that allow `GregorianCalendar` objects to be created using a combination of different date, time zone, and local values. It methods override those provided by the `Calendar` class.

In the following program using the `Calendar` class we measure the time in seconds we need to run two nested `for` loops and then we provide the present time in hours, minutes, seconds, and milliseconds.

```
// MyCalendar.java

import java.util.Calendar;

public class MyCalendar
{
  public static void main(String args[])
  {
  int i, j;
  int t1, t2;

  Calendar time = Calendar.getInstance();
  t1 = time.get(Calendar.SECOND);

  for(i=0;i<40000;i++)
  {
  for(j=0;j<40000;j++)
  {
  j++;
  } // end for loop j
  j = 0;
  } // end for loop i

  time = Calendar.getInstance();
  t2 = time.get(Calendar.SECOND);

  int diff = t2 - t1;

  System.out.println("diff = " + diff);

  int hour = time.get(Calendar.HOUR);
  int minute = time.get(Calendar.MINUTE);
  int second = time.get(Calendar.SECOND);
  int milli = time.get(Calendar.MILLISECOND);
  System.out.println("hour = " + hour);
  System.out.println("minute = " + minute);
  System.out.println("second = " + second);
  System.out.println("millisecond = " + milli);
  } // end main
}
```

We can also use the `Calendar` class to find the age of a person, where the birthdate of the person is given.

```
// Person.java

import java.text.SimpleDateFormat;
import java.text.ParseException;
import java.util.Calendar;
import java.util.GregorianCalendar;

class Person
{
  final String name;
  final GregorianCalendar dob;

  Person(String name,GregorianCalendar dob)
  { this.name = name; this.dob = dob; }

  int age()
  {
  final GregorianCalendar today = new GregorianCalendar();
  final int diff = today.get(Calendar.YEAR) - dob.get(Calendar.YEAR);
  final int this_day = today.get(Calendar.DAY_OF_YEAR);
  final int dob_day = dob.get(Calendar.DAY_OF_YEAR);
  if(this_day > dob_day) { return diff; }
  else { return (diff - 1); }
  }

public static final SimpleDateFormat dmy =
   new SimpleDateFormat("yyyy.mm.dd");

public static GregorianCalendar date(String x) throws ParseException
{
  final GregorianCalendar r = new GregorianCalendar();
  r.setTime(dmy.parse(x));
  return r;
}

public static void main(String[] args) throws ParseException
{
  Person p = new Person("Tibertus Smith",date("1955.03.20"));
  System.out.println("Tibertus Smith age is " + p.age());
}
}
```

To get the current date and time the class `Timestamp` from `java.sql` can be used.

```
// TimeTest.java

import java.sql.Timestamp;
import java.util.Date;

public class TimeTest
{
  public static Timestamp getCurrentTime()
  {
  return new Timestamp(new Date().getTime());
  }

  public static void main(String[] args)
  {
  System.out.println(getCurrentTime());
  }
}
```

The output would like

```
2003-07-16 17:54:27.601
```

4.11 Destroying Objects

Java has no explicit way to destroy an object. However, when there exists no reference to the object, nobody knows where it is and nobody can send messages to this object. The Java run-time system creates a *Garbage Collector* for our application which is a separate execution thread identifying objects which are no longer referenced. Such objects are deleted by the garbage collector. However, before deleting an object, the garbage collector gives the object one more chance to do any clean-up tasks it wishes to do by calling the object's `finalize()` method. This process is known as finalization. All objects inherit this method from the common superclass `Object`. One can explicitly drop an object reference by setting the variable to `null`.

```
// Destroy.java

class Destroy
{
    static public void main(String[] args)
    {
    String s = new String("Egoli");
    System.out.println("s = " + s);  // => Egoli
    s = null;
    System.out.println("s = " + s);  // => null
    s = new String("Gauteng");
    System.out.println("s = " + s);  // => Gauteng
    }
}
```

The next program shows how to use the method `finalize()`.

```
// Final.java

class End
{
  int  myId;
  public End(int sequenceNumber)    { myId = sequenceNumber; }
  public void finalize()    { System.out.println(myId); }
}

class Final
{
   public static void main(String args[])
   {
   End sampleObject;
   for(int k=0;k < 5;k++) sampleObject = new End(k);
   }
}
```

4.12 Regular Expression

A *regular expression* is a string of characters which tells the searcher which string (or strings) we are looking for. The javax.util.regex package is a new package in Java 2 Platform, Standard Edition version 1.4. The package provides a regular expression library. A regular expression is a pattern of characters that describes a set of strings, and is often used in pattern matching. Using the classes in the javax.util.regex package we can match sequences of characters against a regular expression. These classes, which comprise the regular expression library, use the Perl 5 regular expression pattern syntax, and provide a much more powerful way of parsing text than was previously available with the java.io.StreamTokenizer and the java.util.StringTokenizer classes. The regular expression library has three classes:

```
Pattern, Matcher, PatternSyntaxException
```

We have is one class to define the regular expression we want to match (the Pattern), and another class (the Matcher) for searching a pattern in a given string. Most of the work of using the regular expression library is understanding its pattern syntax. The actual parsing is the easy part. The simplest kind of regular expression is a literal. A literal is not simply a character within the regular expression, but a character that is not part of some special grouping or expression within the regular expression. For instance, the literal `"x"` is a regular expression. Using the literal, a matcher, and a string, we can ask "Does the regular expression `'x'` match the entire string?" Here's an expression that asks the question

```
boolean b = Pattern.matches("x",someString);
```

If the pattern `"x"` is the string referenced by `someString`, then `b` is true. Otherwise, `b` is false. The matcher is defined by the `Pattern` class, not the `Matcher` class. The `matches` method is defined by the `Pattern` class as a convenience for when a regular expression is used just once. Normally, we would define a `Pattern` class, a `Matcher` class for the Pattern, and then use the matches method defined by the `Matcher` class

```
Pattern p = new Pattern("x");
Matcher m = p.matcher("sometext");
boolean b = m.matches();
```

Regular expressions can be more complex than literals. Adding to the complexity are wildcards and quantifiers. There is only one wildcard used in regular expressions. It is the period `(.)` character. A wildcard is used to match any single character, possibly even a newline. The quantifier characters are the `+` and `*`. The question mark is also a quantifier character. The `+` character placed after a regular expression allows for a regular expression to be matched one or more times. The `*` is like the `+` character, but works zero or more times. For instance, if we want to find a string with a `j` at the beginning, a `z` at the end, and at least one character between the two,

we use the expression `"j.+z"`. If there doesn't have to be any characters between the `j` and the `z`, we use `"j.*z"` instead. The pattern matching tries to find the largest possible hit within a string. So if we request a match against the pattern `"j.*z"`, using the string `"jazjazjazjaz"`, it returns the entire string, not just a single `"jaz"`. This is called greedy behaviour. It is the default in a regular expression unless we specify otherwise. By placing multiple expressions in parentheses, we can request a match against multi-character patterns. For instance, to match a `j` followed by a `z`, we can use the `"(jz)"` pattern. It is the same as `"jz"`. But, by using parenthesis, we can use the quantifiers and say match any number of `"jz"` patterns: `"(jz)+"`.

Another way of working with patterns is through character classes. With character classes, we specify a range of possible characters instead of specifying individual characters. For instance, if we want to match against any letter from j to z, we specify the range j-z in square brackets: `"[j-z]"`. We could also attach a quantifier to the expression, for example, `"[j-z]+"`, to get an expression matching at least one character between j and z, inclusively. Certain character classes are predefined. These represent classes that are common, and so they have a common shorthand. Some of the predefined character classes are

```
\d    A digit ([0-9])
\D    A non-digit ([^0-9])
\s    A whitespace character [ \t\n\x0B\f\r]
\S    non-whitespace character: [^\s]
\w    A word character: [a-zA-Z_0-9]
\W    A non-word character: [^\w]
```

For character classes, `^` is used for negation of an expression. There is a second set of predefined character classes, called POSIX character classes. These are taken from the POSIX specification, and work with US-ASCII characters only

```
\p{Lower}    A lower-case alphabetic character: [a-z]
\p{Upper}    An upper-case alphabetic character:[A-Z]
\p{ASCII}    All ASCII:[\x00-\x7F]
\p{Alpha}    An alphabetic character:[\p{Lower}\p{Upper}]
\p{Digit}    A decimal digit: [0-9]
\p{Alnum}    An alphanumeric character:[\p{Alpha}\p{Digit}]
\p{Punct}    Punctuation: one of !"#$%&'()*,-./:;<=>?@[\]^_‘{|}~
\p{Graph}    A visible character: [\p{Alnum}\p{Punct}]
\p{Print}    A printable character: [\p{Graph}]
\p{Blank}    A space or a tab: [ \t]
\p{Cntrl}    A control character: [\x00-\x1F\x7F]
\p{XDigit}   A hexadecimal digit: [0-9a-fA-F]
\p{Space}    A whitespace character: [ \t\n\x0B\f\r]
```

The final set of character classes listed here are the boundary matchers. These are meant to match the beginning or end of a sequence of characters, specifically a line, word, or pattern.

```
^     The beginning of a line
$     The end of a line
\b    A word boundary
\B    A non-word boundary
\A    The beginning of the input
\G    The end of the previous match
\Z    The end of the input but for the final terminator, if any
\z    The end of the input
```

The key thing to understand about all the character class expressions is the use of the \. When we compose a regular expression as a Java string, we must escape the \ character. Otherwise, the character following the \ will be treated as special by the javac compiler. To escape the character, specify a double \\. By placing a double \\ in the string, we are saying we want the actual \ character there. For instance, if we want to use a pattern for any string of alphanumeric characters, simply having a string containing `\p{Alnum}*` is not sufficient. We must escape the \ as follows

```
boolean b = Pattern.matches("\\p{Alnum}*",someString);
```

The `Pattern` class is for defining patterns, that is, it defines the regular expression we want to match. Instead of using matches to see if a pattern matches the whole string, what normally happens is we check to see if a pattern matches the next part of the string. To use a pattern we must compile it. The commmand is

```
Pattern pattern = Pattern.compile(somePattern);
```

The `matches` method of the Pattern class compiles the pattern with each call. If we want to use a pattern many times, we can avoid multiple compilation by getting a `Matcher` class for the `Pattern` class and then using the `Matcher` class. After we compile the pattern, we can request to get a `Matcher` for a specific string.

```
Matcher matcher = pattern.matcher(someString);
```

The `Matcher` provides a matches method that checks against the entire string. The class also provides a `find()` method that tries to find the next sequence, possibly not at the beginning of the string, that matches the pattern. After we know we have a match, we can get the match with the `group()` method

```
if(matcher.find()) {
  System.out.println(matcher.group());
}
```

We can also use the matcher as a search and replace mechanism. For instance, to replace all occurrences of a pattern within a string, we use the following expression

```
String newString = matcher.replaceAll("replacement words");
```

All occurrences of the pattern in question would be replaced by the replacement words.

The following program takes three command line arguments. The first argument is a string to search. The second is a pattern for the search. The third is the replacement string. The replacement string replaces each occurrence of the pattern found in the search string.

```
// MyRegExp.java

import java.util.regex.*;

public class MyRegExp
{
  public static void main(String args[])
  {
  if(args.length != 3)
  {
  System.out.println("Use: source string, pattern, replacement string");
  System.exit(-1);
  }

  String source = args[0];
  String thePattern = args[1];
  String replace = args[2];

  Pattern pattern = Pattern.compile(thePattern);
  Matcher match = pattern.matcher(source);
  if(match.find())
  {
  System.out.println(match.replaceAll(replace));
  }
  } // end main
}
```

For example, if we compile the program, and then run it like this

```
java MyRegExp "the cat jumps over the cat" "cat" "dog"
```

It returns:

```
the dog jumps over the dog
```

In next example we replace all small letter in the string `Hello World` by `x`.

```
// MyRegExp1.java

import java.util.regex.*;

public class MyRegExp1
{
  public static void main(String args[])
  {
  String input = new String("Hello World");
  Pattern p = Pattern.compile("[a-z]");
  Matcher m = p.matcher(input);

  if(m.find())
  {
  System.out.println(m.replaceAll("x"));
  }
  } // end main
}
```

The output is

```
Hxxxx Wxxxx
```

4.13 The Bytecode Format

Bytecodes are the machine language of the Java virtual machine (JVM). When a JVM loads a class file, it gets one stream of bytecodes for each method in the class. The bytecodes streams are stored in the method area of the JVM. The bytecodes for a method are executed when that method is invoked during the course of running the program. They can be executed by intepretation, just-in-time compiling, or any other technique that was chosen by the designer of a particular JVM. A method's bytecode stream is a sequence of instructions for the Java virtual machine. Each instruction consists of a one-byte opcode followed by zero or more operands. The opcode indicates the action to take. If more information is required before the JVM can take the action, that information is encoded into one or more operands that immediately follow the opcode. Thus each bytecode consists of a one-byte opcode followed by zero or more bytes of additional operand information. Each type of opcode has a mnemonic. In the typical assembly language style, streams of Java bytecodes can be represented by their mnemonics followed by any operand values.

For example consider the following Java program `Exp1.java` that finds the first 9001 digits of the number e.

```
// Exp1.java

public class Exp1
{
   public static void main(String[] args)
   {
   int N = 9009;
   int n = N;
   int[] a = new int[N];
   int x = 0;

   while((--n != 0))
   a[n] = 1 + 1/n;

   for(;N > 9;System.out.print(x))
   for(n=N--;(--n !=0);a[n]=x%n,x=10*a[n-1]+x/n);
   }
}
```

We compile the Java code with

```
javac Exp1.java
```

to obtain the byte code (class file `Exp1.class`). To disassemble we enter

```
javap -c Exp1 > disassem.txt
```

The contents of the file `disassem.txt` is

```
Compiled from Exp1.java
public class Exp1 extends java.lang.Object {
    public Exp1();
    public static void main(java.lang.String[]);
}

Method Exp1()
   0 aload_0
   1 invokespecial #1 <Method java.lang.Object()>
   4 return

Method void main(java.lang.String[])
   0 sipush 9009   // push two-byte signed integer
   3 istore_1      // store integer in local variable 1
   4 iload_1       // push integer from local variable 1
   5 istore_2      // store integer in local variable 2
   6 iload_1       // push integer in local variable 1
   7 newarray int  // allocate new array for int
   9 astore_3      // store object reference in local variable 3
  10 iconst_0      // push the integer constant 0
  11 istore 4      // store integer in local variable
  13 goto 24       // branch to address
  16 aload_3       // retrieve object reference from local variable 3
  17 iload_2       // push integer from local variable 2
  18 iconst_1      // push integer constant 1
  19 iconst_1      // push integer constant 1
  20 iload_2       // push integer from local variable 2
  21 idiv          // divide one integer by another integer
  22 iadd          // add two integer
  23 iastore       // store in integer array
  24 iinc 2 -1     // increment integer in local variable
  27 iload_2       // push integer from local variable 2
  28 ifne 16       // jump if nonzero
  31 goto 80       // branches to address
  34 iload_1       // pushes integer from local variable 1
  35 dup           // duplicate top single-word item on the stack
  36 iconst_1      // push the integer constant 1
  37 isub          // subtract two integer
  38 istore_1      // store integer in local variable 1
  39 istore_2      // store integer in local variable 2
  40 goto 65       // branches to address
  43 aload_3       // retrieve object-reference from local variable 3
  44 iload_2       // push integer from local variable 2
```

```
45 iload 4       // retrieve integer from local variable
47 iload_2       // push integer from local variable 2
48 irem          // remainder of two integers
49 iastore       // store an integer array
50 bipush 10     // push one-byte signed integer
52 aload_3       // retrieve object reference from local variable 3
53 iload_2       // push integer from local variable 2
54 iconst_1      // push the integer constant 1
55 isub          // subtract two integers
56 iaload        // retrieve integer from array
57 imul          // multiply two integers
58 iload 4       // retrieve integer from local variable
60 iload_2       // pushes integer from local variable 2
61 idiv          // divide an integer by another integer
62 iadd          // add two integers
63 istore 4      // store integer in local variable
65 iinc 2 -1     // increment integer in local variable
68 iload_2       // push integer from local variable 2
69 ifne 43       // jump if nonzero
72 getstatic #2 <Field java.io.PrintStream out>
                 // get value of static field
75 iload 4       // retrieve integer from local variable
77 invokevirtual #3 <Method void print(int)>
                 // call an instance method
80 iload_1       // push integer from local variable 1
81 bipush 9      // push one-byte signed integer
83 if_icmpgt 34  // jump if one integer is greater than another
86 return        // return from method
```

The bytecode instruction can be divided into several categories

Pushing constants onto the stack
Accessing and modifying the value of a register
Accessing arrays
Stack manipulation (`swap`, `dup`, `pop`)
Arithmetic, logical, and conversion instructions
Control transfer
Function return
Manipulating object fields
Method invocation
Object creation
Type casting

With the exception of two table lookup instruction, all instructions are fixed length based on the opcode.

The bytecode instruction set was designed to be compact. All instructions, except two that deal with table jumping, are aligned on byte boundaries. The total number of opcodes is small enough so that opcodes occupy only one byte. This helps minimize the size of class files that may be traveling across networks before being loaded by a JVM. It also helps keep the size of the JVM implementation small. All computation in the JVM centers on the stack. Because the JVM has no registers for storing abitrary values, everything must be pushed onto the stack before it can be used in a calculation. Bytecode instructions therefore operate primarily on the stack. For example,

```
dup
```

duplicates top single-word items on the stack. Thus `dup` pops the top single-word value off te operand stack, and then pushes that value twice - i.e. it makes an extra copy of the top item on the stack. The command

```
ifne <label>
```

pops the top `int` off the operand stack. If the `int` does not equal zero, execution branches to the address (pc + branchoffset), where pc is the address of the `ifne` opcode in the bytecode and branchoffset is a 16-bit signed integer parameter following the `ifne` opcode in the bytecode. If the `int` on the stack equals zero, execution continues at the next instruction. The command

```
if_icmpgt <label>
```

pops the top two ints off the stack and compares them. If value2 is greater than value1, execution branches to the address (pc + branchoffset), where pc is the address of the `if_icmpgt` in the bytecode and branchoffset is a 16-bit signed integer parameter following the `if_icmpgt` opcode in the bytecode. If value2 is less than or equal to value1, execution continues at the next instruction.

Primitive types

As described in chapter 2 the JVM supports seven primitive data types. Java programmers can declare and use variables of these data types, and Java bytecodes operate upon these data types. The seven primitive types are listed in the following table

```
Type   Definition
====   ==========
byte   one-byte signed two's complement integer
short  two-byte signed two's complement integer
int    4-byte signed two's complement integer
long   8-byte signed two's complement integer
float  4-byte IEEE 754 single-precision float
double 8-byte IEEE 754 double-precision float
char   2-byte unsigned Unicode character
```

The primitive types appear as operands in bytecode streams. All primitive types that occupy more than 1 byte are stored in big-endian order in the bytecode stream, which means higher-order bytes precede lower-order bytes. For example, to push the constant value 256 (hex 0100) onto the stack, we would use the `sipush` opcode followed by a `short` operand. The `short` appears in the bytecode stream as 01 00 because the JVM is big-endian. If the JVM were little-endian, the short would appear as 00 01.

Java opcodes generally indicate the type of their operands. This allows operands to just be themselves, with no need to identify their type to the JVM. For example, instead of having one opcode that pushes a local variable onto the stack, the JVM has several. Opcodes `iload`, `lload`, `fload`, and `dload` push local variables of type `int`, `long`, `float`, and `double`, respectively, onto the stack.

Pushing constants onto the stack

Many opcodes push constants onto the stack. Opcodes indicate the constant value to push in three different ways. The constant value is either implicit in the opcode itself, follows the opcode in the bytecode stream as an operand, or is taken from the constant pool. Some opcodes by themselves indicate a type and constant value to push. For example, the `iconst_1` opcode tells the JVM to push integer value one. Such bytecodes are defined for some commonly pushed numbers of various types. These instructions occupy only 1 byte in the bytecode stream. They increase the efficiency of bytecode execution and reduce the size of bytecode streams. The opcodes that push ints and floats are shown in the following table

```
Opcode    Operand(s)  Description
======    ==========  ===========
iconst_m1 none        pushes int -1 onto the stack
iconst_0  none        pushes int 0 onto the stack
iconst_1  none        pushes int 1 onto the stack
iconst_2  none        pushes int 2 onto the stack
iconst_3  none        pushes int 3 onto the stack
iconst_4  none        pushes int 4 onto the stack
iconst_5  none        pushes int 5 onto the stack
fconst_0  none        pushes float 0 onto the stack
fconst_1  none        pushes float 1 onto the stack
fconst_2  none        pushes float 2 onto the stack
```

The opcodes shown in the previous table push ints and floats, which are 32-bit values. Each slot on the Java stack is 32 bits wide. Therefore each time an `int` or `float` is pushed onto the stack, it occupies one slot. The opcodes shown in the next table push longs and doubles. `Long` and `double` values occupy 64 bits. Each time a `long` or `double` is pushed onto the stack, its value occupies two slots on the stack. Opcodes that indicate a specific `long` or `double` value to push are shown in the following table:

```
Opcode    Operand(s)  Description
======    ==========  ===========
lconst_0  none        pushes long 0 onto the stack
lconst_1  none        pushes long 1 onto the stack
dconst_0  none        pushes double 0 onto the stack
dconst_1  none        pushes double 1 onto the stack
```

One other opcode pushes an implicit constant value onto the stack. The `aconst_null` opcode, shown in the following table, pushes a `null` object reference onto the stack. The format of an object reference depends upon the JVM implementation. An object reference will somehow refer to a Java object on the garbage-collected heap. A `null` object reference indicates an object reference variable does not currently refer to any valid object. The `aconst_null` opcode is used in the process of assigning `null` to an object reference variable.

```
Opcode       Operand(s)  Description
==========   ==========  ===========
aconst_null  none        pushes a null object reference onto the stack
```

Two opcodes indicate the constant to push with an operand that immediately follows the opcode. These opcodes, shown in the following table, are used to push integer constants that are within the valid range for byte or short types. The `byte` or `short` that follows the opcode is expanded to an `int` before it is pushed onto the stack, because every slot on the Java stack is 32 bits wide. Operations on bytes and shorts that have been pushed onto the stack are actually done on their int equivalents.

```
Opcode  Operand(s)    Description
======  ==========    ===========
bipush  byte1         expands byte1 (a byte type) to
                      an int and pushes it onto the stack
sipush  byte1, byte2  expands byte1, byte2 (a short type)
                      to an int and pushes it onto the stack
```

Three opcodes push constants from the constant pool. All constants associated with a class, such as final variables values, are stored in the class's constant pool. Opcodes that push constants from the constant pool have operands that indicate which constant to push by specifying a constant pool index. The Java virtual machine will look up the constant given the index, determine the constant's type, and push it onto the stack. The constant pool index is an unsigned value that immediately follows the opcode in the bytecode stream. Opcodes `lcd1` and `lcd2` push a 32-bit item onto the stack, such as an `int` or `float`. The difference between `lcd1` and `lcd2` is that `lcd1` can only refer to constant pool locations one through 255 because its index is just 1 byte. Constant pool location zero is unused. `lcd2` has a 2-byte index, so it can refer to any constant pool location. `lcd2w` also has a 2-byte index, and it is used to refer to any constant pool location containing a `long` or `double`, which occupy 64 bits. The opcodes that push constants from the constant pool are shown in the following table

Opcode	Operand(s)	Description
ldc1	indexbyte1	pushes 32-bit constant_pool entry specified by indexbyte1 onto the stack
ldc2	indexbyte1, indexbyte2	pushes 32-bit constant_pool entry specified by indexbyte1, indexbyte2 onto the stack
ldc2w	indexbyte1, indexbyte2	pushes 64-bit constant_pool entry specified by indexbyte1, indexbyte2 onto the stack

Pushing local variables onto the stack

Local variables are stored in a special section of the stack frame. The stack frame is the portion of the stack being used by the currently executing method. Each stack frame consists of three sections – the local variables, the execution environment, and the operand stack. Pushing a local variable onto the stack actually involves moving a value from the local variables section of the stack frame to the operand section. The operand section of the currently executing method is always the top of the stack, so pushing a value onto the operand section of the current stack frame is the same as pushing a value onto the top of the stack. The Java stack is a last-in, first-out stack of 32-bit slots. Because each slot in the stack occupies 32 bits, all local variables occupy at least 32 bits. Local variables of type `long` and `double`, which are 64-bit quantities, occupy two slots on the stack. Local variables of type `byte` or `short` are stored as local variables of type `int`, but with a value that is valid for the smaller type. For example, an `int` local variable which represents a `byte` type will always contain a value valid for a `byte` ($-128 \leq value \leq 127$).

Each local variable of a method has a unique index. The local variable section of a method's stack frame can be thought of as an array of 32-bit slots, each one addressable by the array index. Local variables of type `long` or `double`, which occupy two slots, are referred to by the lower of the two slot indexes. For example, a `double` that occupies slots two and three would be referred to by an index of two. Several opcodes exist that push `int` and `float` local variables onto the operand stack. Some opcodes are defined that implicitly refer to a commonly used local variable position. For example, `iload_0` loads the `int` local variable at position zero. Other local variables are pushed onto the stack by an opcode that takes the local variable index from the first byte following the opcode. The `iload` instruction is an example of this type of opcode. The first byte following `iload` is interpreted as an unsigned 8-bit index that refers to a local variable. Unsigned 8-bit local variable indexes, such as the one that follows the `iload` instruction, limit the number of local variables in a method to 256. A separate instruction, called `wide`, can extend an 8-bit index by another 8 bits. This raises the local variable limit to 64 kilobytes. The `wide` opcode is followed by an 8-bit operand. The `wide` opcode and its operand

can precede an instruction, such as `iload`, that takes an 8-bit unsigned local variable index. The JVM combines the 8-bit operand of the `wide` instruction with the 8-bit operand of the `iload` instruction to yield a 16-bit unsigned local variable index.

The opcodes that push int and float local variables onto the stack are shown in the following table:

```
Opcode  Operand(s)  Description
======  ==========  ===========
iload    vindex     pushes int from local variable position vindex
iload_0  none       pushes int from local variable position zero
iload_1  none       pushes int from local variable position one
iload_2  none       pushes int from local variable position two
iload_3  none       pushes int from local variable position three
fload    vindex     pushes float from local variable position vindex
fload_0  none       pushes float from local variable position zero
fload_1  none       pushes float from local variable position one
fload_2  none       pushes float from local variable position two
fload_3  none       pushes float from local variable position three
```

The next table shows the instructions that push local variables of type long and double onto the stack. These instructions move 64 bits from the local variable section of the stack frame to the operand section.

```
Opcode  Operand(s)  Description
======  ==========  ===========
lload   vindex      pushes long from local variable positions vindex
                    and vindex+1
lload_0 none        pushes long from local variable positions zero
                    and one
lload_1 none        pushes long from local variable positions one
                    and two
lload_2 none        pushes long from local variable positions two
                    and three
lload_3 none        pushes long from local variable positions three
                    and four
dload   vindex      pushes double from local variable positions vindex
                    and vindex+1
dload_0 none        pushes double from local variable positions zero
                    and one
dload_1 none        pushes double from local variable positions one
                    and two
dload_2 none        pushes double from local variable positions two
                    and three
dload_3 none        pushes double from local variable positions three
                    and four
```

The final group of opcodes that push local variables move 32-bit object references from the local variables section of the stack frame to the operand section. These opcodes are shown in the following table:

```
Opcode  Operand(s)  Description
======  ==========  ===========
aload   vindex      pushes object reference from local variable
                    position vindex
aload_0 none        pushes object reference from local variable
                    position zero
aload_1 none        pushes object reference from local variable
                    position one
aload_2 none        pushes object reference from local variable
                    position two
aload_3 none        pushes object reference from local variable
                    position three
```

Popping to local variables

For each opcode that pushes a local variable onto the stack there exists a corresponding opcode that pops the top of the stack back into the local variable. The names of these opcodes can be formed by replacing `load` in the names of the push opcodes with store. The opcodes that pop ints and floats from the top of the operand stack to a local variable are listed in the following table. Each of these opcodes moves one 32-bit value from the top of the stack to a local variable.

```
Opcode   Operand(s)  Description
======   ==========  ===========
istore   vindex      pops int to local variable position vindex
istore_0 none        pops int to local variable position zero
istore_1 none        pops int to local variable position one
istore_2 none        pops int to local variable position two
istore_3 none        pops int to local variable position three
fstore   vindex      pops float to local variable position vindex
fstore_0 none        pops float to local variable position zero
fstore_1 none        pops float to local variable position one
fstore_2 none        pops float to local variable position two
fstore_3 none        pops float to local variable position three
```

The next table shows the instructions that pop values of type long and double into a local variable. These instructions move a 64-bit value from the top of the operand stack to a local variable.

```
Opcode   Operand(s)  Description
======   ==========  ===========
lstore   vindex      pops long to local variable positions vindex
```

```
                    and vindex+1
lstore_0 none       pops long to local variable positions zero and one
lstore_1 none       pops long to local variable positions one and two
lstore_2 none       pops long to local variable positions two and three
lstore_3 none       pops long to local variable positions three and four
dstore   vindex     pops double to local variable positions vindex
                    and vindex+1
dstore_0 none       pops double to local variable positions zero and one
dstore_1 none       pops double to local variable positions one and two
dstore_2 none       pops double to local variable positions two and three
dstore_3 none       pops double to local variable positions three and four
```

The final group of opcodes that pops to local variables are shown in the following table. These opcodes pop a 32-bit object reference from the top of the operand stack to a local variable.

```
Opcode  Operand(s)  Description
======  ==========  ===========
astore    vindex    pops object reference to local variable position vindex
astore_0  none      pops object reference to local variable position zero
astore_1  none      pops object reference to local variable position one
astore_2  none      pops object reference to local variable position two
astore_3  none      pops object reference to local variable position three
```

Type conversions

The Java virtual machine has many opcodes that convert from one primitive type to another. No opcodes follow the conversion opcodes in the bytecode stream. The value to convert is taken from the top of the stack. The JVM pops the value at the top of the stack, converts it, and pushes the result back onto the stack. Opcodes that convert between `int`, `long`, `float`, and `double` are shown in the following table. There is an opcode for each possible from-to combination of these four types:

```
Opcode  Operand(s)  Description
======  ==========  ===========
i2l     none        converts int to long
i2f     none        converts int to float
i2d     none        converts int to double
l2i     none        converts long to int
l2f     none        converts long to float
l2d     none        converts long to double
f2i     none        converts float to int
f2l     none        converts float to long
f2d     none        converts float to double
d2i     none        converts double to int
d2l     none        converts double to long
d2f     none        converts double to float
```

Opcodes that convert from an int to a type smaller than `int` are shown in the following table. No opcodes exist that convert directly from a `long`, `float`, or `double` to the types smaller than `int`. Therefore converting from a `float` to a `byte`, for example, would require two steps. First the `float` must be converted to an int with `f2i`, then the resulting `int` can be converted to a `byte` with `int2byte`.

Opcode	Operand(s)	Description
int2byte	none	converts int to byte
int2char	none	converts int to char
int2short	none	converts int to short

Although opcodes exist that convert an `int` to primitive types smaller than `int` (`byte`, `short`, and `char`), no opcodes exist that convert in the opposite direction. This is because any bytes, shorts, or chars are effectively converted to `int` before being pushed onto the stack. Arithmetic operations upon bytes, shorts, and chars are done by first converting the values to int, performing the arithmetic operations on the ints, and being happy with an int result.

The maximum value for a byte is 127. The minimum value is −128. Values of type `int` that are within this range convert directly to `byte`. Consider now the case, when the int gets beyond the valid range for byte, things get interesting. The JVM converts an `int` to a `byte` by truncating and sign extending. The highest order bit, the sign bit, of longs, ints, shorts, and bytes indicate whether or not the integer value is positive or negative. If the sign bit is zero, the value is positive. If the sign bit is one, the value is negative. Bit 7 of a byte value is its sign bit. To convert an `int` to a `byte`, bit 7 of the `int` is copied to bits 8 through 31. This produces an `int` that has the same numerical value that the int's lowest order byte would have if it were interpreted as a byte type. After the truncation and sign extension, the int will contain a valid byte value. What happens when an `int` that is just beyond the valid range for byte types gets converted to a `byte`. For example, when the variable has a value of 128 (0x00000080) and is converted to `byte`, the resulting byte value is −128 (0xffffff80). Later, when the variable has a value of −129 (0xffffff7f) and is converted to byte, the resulting byte value is 127 (0x0000007f).

Chapter 5

Inheritance and Abstract Classes

5.1 Introduction

Inheritance is an integral part of Java and any OOP language such as C++. In Java we are always using inheritance when we create a class, because if we do not override we inherit from Java's standard root `class Object`. When we use inheritance we say

The derived class is like the base class

with some additional information added to the derived class. In the class the keyword is `extends` followed by the name of the base class. Inheritance does not just copy the interface of the base class. When we create an object of the derived class it contains within it a subobject of the base class. This subobject is the same as if we had created an object of the base class itself. It is important that the base class is initialized correctly. We are always doing inheritance when we create a class, because if we do not say otherwise we inherit from Java's root class `Object`. Java does not support multiple inheritance. Thus a class can only have one super class.

An *abstract class* is a class that has at least one abstract method. An abstract method is a method that is declared with only its signature; it has no implementation. Since it has at least one abstract method, an abstract class cannot be instantiated. Classes and methods are declared to be abstract by means of the `abstract` modifier. A number of Java classes are declared `abstract`. For example

```
public abstract class AbstractList
extends AbstractCollection
implements List
```

We also allowed to write our own abstract class and then in the derived classes give the implementations.

In the following section we give examples for the use of abstract classes and inheritance.

5.2 Abstract Class

In the following program the `class Circle` and the `class Square` implements the abstract method

```
abstract double area();
```

An abstract method can be viewed as an outline or a specification contract. It specifies what its subclasses have to implement, but leaves the actual implementation up to them. The abstract `area()` method is declared in the `Shape class`, because we want every subclass to have a complete method that returns the areas of its instances, and we want all those methods to have the same signature `double area()`. The abstract method in the abstract superclass enforces that specification.

```
// MTest.java

import java.math.*;

abstract class Shape
{
   abstract double area();
}

class Circle extends Shape
{
   private double radius;

   Circle(double radius)    // constructor
   {
   this.radius = radius;
   }

   double area()
   {
   return Math.PI*radius*radius;
   }

}   // end class Circle

class Square extends Shape
{
   private double side;

   Square(double side)   // constructor
   {
   this.side = side;
```

```
   }

   double area()
   {
   return side*side;
   }

}  // end class Square

class Triangle extends Shape
{
   private double base, altitude;

   Triangle(double base,double altitude)  // constructor
   {
   this.base = base;
   this.altitude = altitude;
   }

   double area()
   {
   return 0.5*base*altitude;
   }

}  // end class Triangle


class MTest
{
   public static void main(String[] args)
   {
   Circle circle = new Circle(2.5);
   System.out.println("area of circle is: " + circle.area());

   Square square = new Square(3.1);
   System.out.println("area of square is: " + square.area());

   Triangle triangle = new Triangle(2.0,5.0);
   System.out.println("area of triangle is: " + triangle.area());
   }
}
```

5.3 Inheritance

Inheritance is the creation of one class by extending another class so that instances of the new class automatically inherit the fields and methods of its parent class. The keyword is `extends`. In the following program the line

```
class ClassB extends ClassA
```

defines `class ClassB` to be a subclass of `class ClassA`. We override `toString()`.

```
// Inher.java

class ClassA
{
   protected double x;

   public String toString()
   {
   x = 4.5;
   return new String("(" + x + ")");
   }
}

class ClassB extends ClassA
{
   private double y;

   public String toString()
   {
   y = 1.5;
   return new String("(" + x + "," + y + ")");
   }
}

class Inher
{
   public static void main(String[] args)
   {
   ClassA a = new ClassA();
   System.out.println("a = " + a);
   ClassB b = new ClassB();
   System.out.println("b = " + b);
   }
}
// output
// (4.5)
// (0.0,1.5)
```

The following program shows how a subclass's fields and methods override those of its superclass. In the following program we override the methods `void g()` and `String toString()`.

```
// Inher1.java

class ClassA
{
   protected int m;
   protected int n;

   void f()
   {
   System.out.println("In ClassA method f() ");
   m = 15;
   }

   void g()
   {
   System.out.println("In ClassA method g() ");
   n = 31;
   }

   public String toString()
   {
   return new String("{ m=" + m + ", n=" + n  + " }" );
   }
}  // end ClassA

class ClassB extends ClassA
{
   private double n;

   void g()
   {
   System.out.println("In ClassB method g() ");
   n = 2.789;
   }

   public String toString()
   {
   return new String("{ m=" + m + ", n=" + n  + " }" );
   }
}  // end ClassB
```

```
class Inher1
{
    public static void main(String[] args)
    {
    ClassA a = new ClassA();
    a.f();
    a.g();
    System.out.println("a = " + a);

    ClassB b = new ClassB();
    b.f();
    b.g();
    System.out.println("b = " + b);
    }
}
```

The output is

```
In ClassA method f()
In ClassA method g()
a = { m=15, n=31 }
In ClassA method f()
In ClassB method g()
b = { m=15, n=2.789 }
```

5.4 Composition

Inheritance uses a - *is a relationship* -. Composition is the creation of one class using another class for its component data. It uses the - *has a relationship* -. For example a person has a name. It is very common to use composition and inheritance together. The following program shows such a case. Both composition and inheritance allow to place subobjects inside our new class. The `protected` keyword says "this is `private` as far as the class user is concerned", but available to anyone who inherits from this class.

Java uses the keyword `super` to refer to members of the parent class. When used in the form `super()` it invokes the superclass's constructor. When used in the form `super.f()` it invokes the function `f()` declared in the superclass. Thus we can override the override. The method `String trim()` in the `String` class removes white space from both ends of this string.

```
// MInher.java

class Name
{
  private String first;
  private String middle;
  private String last;

  Name() { }

  Name(String first, String last)
  { this.first = first;   this.last = last;  }

  Name(String first, String middle, String last)
  { this(first,last);  this.middle = middle;  }

  String first() { return first;  }

  String middle() { return middle;  }

  String last() { return last;  }

  void setFirst(String first) { this.first = first;  }

  void setMiddle(String middle) { this.middle = middle;  }

  void setLast(String last) { this.last = last;  }

  public String toString()
```

```
  {
  String s = new String();
  if(first != null) s+= first + " ";
  if(middle != null) s += middle + " ";
  if(last != null) s += last + " ";
  return s.trim();
  }
}

class Person
{
  protected Name name;
  protected char sex;    // 'M' or 'F'
  protected String id;   // e.g., social Security number

  Person(Name name,char sex)
  { this.name = name; this.sex = sex; }

  Person(Name name,char sex,String id)
  { this.name = name;  this.sex = sex;  this.id = id; }

  Name name() { return name;  }

  char sex() { return sex;  }

  String id() { return id;  }

  void setId(String id) { this.id = id;  }

  public String toString()
  {
  String s = new String(name + " (sex: " + sex);
  if(id != null) s += "; id: " + id;
  s += ")";
  return s;
  }
}

class Student extends Person
{
  protected int credits;     // credit hours earned
  protected double gpa;      // grad-point average

  Student(Name name,char sex,int credits,double gpa)
  {
```

```
  super(name,sex);
  this.credits = credits;
  this.gpa = gpa;
  }

  int credits() { return credits;  }

  double gpa() { return gpa;  }

  public String toString()
  {
  String s;
  s = new String(super.toString());    // invokes Person.toString()
  s += "\n\tcredits: " + credits;
  s += "\n\tgpa:     " + gpa;
  return s;
  }
}

class MInher
{
  public static void main(String[] args)
  {
  Name fc = new Name("Francis", "Harry Compton", "Crick");
  System.out.println(fc + " won the 1962 Nobel in Physiology.");
  System.out.println("His first name was " + fc.first());

  Name bobsName = new Name("Robert", "LKee");
  Person bob = new Person (bobsName, 'M');
  System.out.println("bob: " + bob);
  bob.name.setMiddle("Edward");
  System.out.println("bob: " + bob);
  Person ann = new Person(new Name("Ann", "Baker"), 'F');
  System.out.println("ann: " + ann);
  ann.setId("0530117366");
  System.out.println("ann: " + ann);

  Name rolandsName = new Name("Roland", "Flechter");
  Student roland = new Student(rolandsName,'M',16,3.5);
  System.out.println("Roland: " + roland);
  }
}
```

5.5 Constructors

When an object is created, its member can be initialized by a constructor method. A *constructor* is not a method but a special operator which is executed when a new instance of a class is created. Depending on the arguments passed to the class when the instance is generated, a different constructor can be called. By default, every class has a single constructor with no parameters the so-called *default constructor*. Besides the default constructor we can define other constructors. We note that constructors, unlike methods, are not inherited.

In the following program we provide for the `class MPoint` a default constructor and a constructor which takes as argument two `double`. We override the methods `toString()` and `equals()` from the `Object` class.

```
// MPoint.java

public class MPoint
{
  private double x;
  private double y;

  public MPoint()          // default constructor
  {
    x = 1.0; y = 0.0;
  }

  public MPoint(double a,double b)  // constructor
  {
    x = a; y = b;
  }

  public double square_length(MPoint p)  // method
  {
    return p.x*p.x + p.y*p.y;
  }

  public double square_distance(MPoint p,MPoint q)  // method
  {
    return (p.x - q.x)*(p.x - q.x) + (p.y - q.y)*(p.y - q.y);
  }

  public boolean equals(MPoint p,MPoint q)  // overriden method
  {
  if((p.x == q.x) && (p.y == q.y))
  return true;
```

```
  else
  return false;
  }

  public String toString()  // overriden method
  {
  return new String("(" + x + ", " + y + ")");
  }

  public static void main(String[] args)
  {
    MPoint P;
    P = new MPoint(4.5,5.5);
    MPoint Q = new MPoint(3.0,1.0);

    double length_square = P.square_length(P);
    double distance_square = P.square_distance(P,Q);
    System.out.print("Length of P squared = ");
    System.out.println(length_square);
    System.out.print("Distance between P and Q squared = ");
    System.out.println(distance_square);

    boolean b = P.equals(Q);
    System.out.println("b = " + b);

    String st = P.toString();
    System.out.println("st = " + st);

    MPoint access;                // default constructor called
    access = new MPoint();
    double result = access.x;
    System.out.println(result);   // => 1.0
  }
}
```

The above program the `class MPoint` contains the `main` method. In the following program we have two classes in one file. The `class MMain` contains the `main` method. The `class MPoint` is standealone. In order that we can use constructors, methods and data members of `class MPoint` in `class MMain` they have to be declared `public`.

```
// MMain.java

class MPoint
{
   private double x;
   private double y;

   public double z;   // public data member

   public MPoint()    // default constructor
   {
   x = 0.0;  y = 0.0;  z = 1.0;
   }

   public MPoint(double a,double b)  // constructor
   {
   x = a;  y = b;
   }

   public double square_length(MPoint p)  // method
   {
   z = p.x*p.x + p.y*p.y;
   return p.x*p.x + p.y*p.y;
   }

   public boolean equals(MPoint p,MPoint q)  // method
   {
   if((p.x == q.x) && (p.y == q.y))
   return true;
   else
   return false;
   }

   public String toString()  // overriden method
   {
   return new String("(" + x + ", " + y + ")");
   }
}  // end class MPoint
```

```
class MMain
{
   public static void main(String[] args)
   {
   MPoint P;
   P = new MPoint(4.5,5.5);
   double distance_square = P.square_length(P);
   System.out.println(distance_square); // 50.5

   double value = P.z;
   System.out.println(value);           // 50.5
   }
}
```

If we change the line in the above program

```
public double z;
```

in the class `MPoint` to

```
private double z;
```

then we obtain the error message

```
variable z in class MPoint is not accessible from class MMain
```

After compiling the program with

```
javac MMain.java
```

we obtain two class files

```
MPoint.class   MMain.class
```

Next we put the two classes into two different files. The file `MMPoint.java` contains the `class MMPoint`. It does not contain a `main` method. To use it we have to compile it using `javac MMPoint.java`. This provides us with the class file `MMPoint.class`. The file `MMain.java` contains the `main` method.

```
// MMPoint.java

public class MMPoint
{
   private double x;
   private double y;
   public double z;

   public MMPoint()
   {
   x = 0.0;
   y = 0.0;
   z = 0.0;
   }

   public MMPoint(double a,double b)
   {
   x = a;  y = b;
   }

   public double square_length(MMPoint p)
   {
   z = p.x*p.x + p.y*p.y;
   return p.x*p.x + p.y*p.y;
   }

   public boolean equals(MPoint p,MPoint q)
   {
   if((p.x == q.x) && (p.y == q.y))
   return true;
   else
   return false;
   }

   public String toString()
   {
   return new String("(" + x + ", " + y + ")");
   }
}
```

The `class MMMain` calls the `class MMPoint` which is in a separate file. To run the program we first have to compile the file `MMPoint.java`, i.e.

```
javac MMPoint.java
```

to obtain the file

```
MMPoint.class
```

Then we compile file `MMMain.java` by entering the commmand

```
javac MMMain.java
```

To run the program we enter

```
java MMMain
```

The file `MMMain.java` only includes the `class MMMain` with the `main` method. It uses the constructor

```
MMPoint(double,double)
```

and the method

```
square_length(MMPoint)
```

of the `class MMPoint`. Furthermore we access the public data member `z` of class `MMPoint`. This can be done, because `double z` is declared `public`.

```
// MMMain.java

public class MMMain
{
   public static void main(String[] args)
   {
   MMPoint P;
   P = new MMPoint(4.5,5.5);
   double distance_square = P.square_length(P);
   System.out.println(distance_square);         // 50.5

   double value = P.z;
   System.out.println(value);                   // 50.5
   }
}
```

5.6 Inner Classes

A feature added to JDK 1.1 are *inner classes*. Inner classes are defined within the scope of an outer (or top-level) class. The inner class is a useful feature because it allows to group classes that logically belong together and control the visibility of one within the other. These inner classes can be defined within a block of statements or (anonymously) within an expression. The following program gives an application of this concept.

```
// Employee.java

public class Employee
{
   int age = 0;
   public String name = "Charles";
   double rate = 111.34;
   Address address;
   Wage wage;

   public Employee(String aName,int number, String aStreet,
                   String aCity,double ratePerHour,int hours)
   {
   name = aName;
   rate = ratePerHour;
   address = new Address(number,aStreet,aCity);
   wage = new Wage(hours);
   }

   // inner class
   class Address
   {
   int number = 0;
   String street = "";
   String city = "";

   Address(int num,String aStreet,String aCity)
   {
   number = num;
   street = aStreet;
   city = aCity;
   }

   void printDetails()
   {
```

```
   System.out.println(number + " " + street + " , " + city);
   }

   } // end class Address

   // inner class
   class Wage
   {
   int hoursWorked = 0;

   Wage(int hours)
   {
   hoursWorked = hours;
   }

   void printDetails()
   {
   System.out.println("Pay packet = " + hoursWorked*rate);
   }

   } // end class Wage

   public static void main(String[] args)
   {
   Employee[] e = new Employee[2];
   e[0] = new Employee("Bob",33,"Lane Street","Springfield",4.5,56);
   e[1] = new Employee("Lucky",123,"Pop Street","Lancing",5.7,34);

   e[0].printInformation();
   e[1].printInformation();
   }

   public void printInformation()
   {
   System.out.println("\nFor Employee: " + name);
   address.printDetails();
   wage.printDetails();
   }
}
```

5.7 Interfaces

Subclasses are used to redefine the behaviour and data structures of a superclass. As mentioned before Java supports single inheritance while C++ supports multiple inheritance. Multiple inheritance is where a subclass can inherit from more than one superclass. However, problems can arise when attempting to determine which methods are executed. Java introduces the concept of *interfaces* to overcome one of the most significant problems with single inheritance. Thus the `interface` keyword takes the abstract concept one step further. Thus the `interface` is used to establish a protocol between classes. The Java interface construct is essentially a skeleton which specifies the protocol that a class must provide if it implements that interface. That is, it indicates which must be available from any class which implements that interface. The `interface` itself does not define any functionality. The format for an `interface` is

```
access-modifier interface name
{
   static variable definitions
   method headers ...
}
```

For example

```
// ColorConstants.java

public interface ColorConstants
{
   int red = 0xff0000; int green = 0x00ff00; int blue = 0x0000ff;
}
```

We compile the file `ColorConstants.java` to obtain the class file. We can now apply the interface with the following program

```
// ColorImpl.java

public class ColorImpl implements ColorConstants
{
   public static void main(String[] args)
   {
   int myblue = ColorConstants.blue;
   System.out.println("myblue = " + myblue); // => 255
   }
}
```

Chapter 6

The GUI and its Components

6.1 Introduction

The AWT (Abstract Window Tookit) allows Java developers to quickly build Java applets and applications using a group of prebuilt user interface components. A number of Java IDE's are available which support the creation of user interfaces using the AWT by dragging-and-dropping components off a toolbar. These IDE'S actually generate Java code on the fly based on what we do in the graphical design environment. This is in contrast to toolset such as Microsoft Visual Basic which separate user interface design from the application code. The advantage of the Java approach is that we can edit our GUI either through a graphical IDE or by simple modifying the Java code and recompiling.

The three steps common to all Java GUI applications are:

1. Creation of a container

2. Layout of GUI components

3. Handling of events

This container object is actually derived from the `java.awt.Container` class and is one of (or inherited from) three primary classes:

`java.awt.Window, java.awt.Panel, java.awt.ScrollPane .`

The `Window` class represents a standalone window (either an application window in the form of a `java.awt.Frame`, or a dialog dox in the form of a `java.awt.Dialog`).

The `java.awt.Panel` class is not a standalone window in and of itself; instead, it acts as a backgroud container for all other components on a form. for instance, the `java.awt.Applet` class is a derect descendant of `java.awt.Panel`.

The main primitive objects derived from class `Component` are

`Button, Checkbox, Label, List, ScrollBar, TextArea, TextField`

Container objects are derived from `Component`. The main ones derived from that are

`Dialog, FileDialog, Frame, Panel, Window`

GUI components can be arranged on a container using one of two methords. the first method involves the exact positioning (by pixel) of components on the container. the second method involves the use of what Java calls *Layout Managers*. If we think about it, vitually all GUIs arrange components based on the row-column metaphor. In other words, buttons, text boxes, and list generally are not located at random all over a form. Instead, they are usually neatly arranged in rows or columns with an OK or Cancel button arranged at the bottom or on the side. Layout Managers allow us to quickly add components to the manager and then have it arrange them automatically. The AWT provides six layout managers for our use:

1. `java.AWT.BorderLayout`
2. `java.awt.FlowLayout`
3. `java.awt.CardLayout`
4. `java.awt.GridLayout`
5. `java.awt.GridBagLayout`
6. `java.awt.BoxLayout`

The `Container` class contains the `setLayout()` method so that we can set the default LayoutManager to be used by our GUI. To actually add components to the container, we use the container's `add()` method.

All events generated are subclasses of `java.awt.AWTEvent`, and contain information such as the position where the event was caused and the source (the class where the event is generated). Different event classes can be used, for example a `MouseEvent` is generated when a mouse is used. If a programmer is only interested in the mouse when it is used to press a button (and not in whether it is the left or right mouse button that has been pressed), they can simply use an `ActionEvent`.

A piece of code that needs to know if an event happens should implement an `EventListener` and and register it with components that may generate an event. If an event is generated the listener is called with the event as a parameter.

The `class Button` provides buttons. Buttons are for clicking. We can specify their labels and control what happens when they are clicked.

A `Checkbox` is a graphical component that can be in either an "on" (true) or "off" (false) state. Clicking on a `Checkbox` changes it state from "on" to "off" or from "off" to "on". With a `Checkbox` any number can be selelected at the same time. The constructor `Checkbox(String label)` creates a checkbox with the specified label.

The `Choice` class presents a pop-up menu of choices. The current choice is displayed as the title of the menu. Thus their are drop-down lists from which the user can select one option.

A `Label` object is a component for placing text in a container. A label displays a single line of read-only text. Labels are purely for display. We need them because there is no reliable way to integrate `drawString` items with GUI components. The `drawString` items have their positions set by co-ordinates, while GUI components are placed whereever they will fit with Layouts giving us some control.

The `List` component presents the user with a scrolling list of text items. The list can be set up so that the user can choose either one item or multiple items.

A `TextField` object is a text component that allows for the editing of a single line of text. The constructor `Textfield(String text,int cols)` constructos a new `Textfield`, `text` is the text to be displayed and `cols` is the number of columns.

A `TextArea` object is a multiline region that displays text. It can be set to allow editing or to be read-only. With the `TextArea` constructor we can control whether a `TextArea` will have scroll bars: vertical, horizontal, both or neither.

6.2 Class Component and Class Container

A component is an object having a graphical representation that can be displayed on the screen and that can interact with the user (for example a button or a textfield). The generic Abstract Window Toolkit (AWT) container object is a component that can contain other AWT components. Components added to a container are tracked in a list. The method

```
void add(Component)   // class Container
```

in class `Container` adds the specified component to the end of the container. The component could be for example a `Button` or a `TextField`.

The method

```
void setSize(int width,int height) // class Component
```

resizes this component so that it has width `width` and height `height`. The method

```
addWindowListener(this) // class Window
```

adds the specified window listener to receive window events from this window. In a method, the keyword `this` refers to the object on which the method operates. The method

```
void setTitle(String title) // class Frame
```

sets the title for the frame of the specified title.

Layout managers help to produce portable, presentable user interfaces. There are a number of different layout managers which use different philosophies to handle the way in which they lay out components. They are

```
FlowLayout, BorderLayout, GridLayout, GridBagLayout, CardLayout
```

The constructor `FlowLayout()` in the `class FlowLayout` constructs a new flow layout. The `FlowLayout` manager is the simplest manager. It lays the components out in a row across a window. When there is no more space left, it starts on the next row, and so until no more space is left. The `BorderLayout` class breaks up the available space on the container component into five regions, north, south, east, west and centre. When adding a component to a border layout we have to specify which region it will go. There are two constructors for the `BorderLayout` class. The constructor

```
BorderLayout()
```

creates a new `BorderLayout` with the default horizontal and vertical gaps of zero. The constructor

```
BorderLayout(int hor,int ver)
```

creates a new `BoderLayout` with the specified horizontal and vertical gaps between the components. The `BorderLayout` class contains the following data fields

```
static String CENTER  (middle of container)
static String EAST    (right side of container)
static String NORTH   (top of container)
static String SOUTH   (bottom of container)
static Strig  WEST    (left side of container)
```

The `GridLayout` class divides the container area into a rectangular grid. Every new compnent we add goes into a single grid cell. The cells are filled from left to right and top to bottom. The `GridLayout()` constructor creates a grid layout with a default of one column per component, in a single row. The constructor

```
GridLayout(int rows,int cols,int hgap,int vgap)
```

creates a grid layout with the specified number of columns and rows. This class is a flexible layout manager that aligns components vertically and horizontally without requiring that the components be the same size. The `CardLayout` object is a layout manager for a container. It treats each component in the container as a card. Only one card is visible at the time, and the containter acts as a stack of cards.

The method

```
void setLayout(LayoutManger mgr) // class Container
```

sets the layout manger for this container.

The `class WindowListener` includes the methods

```
windowClosing, windowClosed, windowOpened, windowDeiconified
```

```
windowIconified, windowActivated, windowDeactivated
```

The method `windowClosing` is invoked when a window is in the process of being closed. It uses the method `exit()` from the `System` class.

The method

```
void setVisible(boolean b) // class Component
```

shows or hides this component depending on the value of parameter `b`.

The following two programs show an application of the classes `Container` and `Component` togther with the classes `Button`, `Checkbox`, `Choice`, `Label`, `List` and `TextField`.

```
// GUI.java

import java.awt.*;
import java.awt.event.*;

class GUI extends Frame
{
   public GUI() // default constructor
   {
   setSize(200,200);
   addWindowListener(new WindowAdapter()
   { public void windowClosing(WindowEvent event)
   { System.exit(0); }});

   setTitle("GUI and its component");
   setLayout(new FlowLayout());

   Button b1 = new Button("Button 1");
   Button b2 = new Button("Button 2");
   add(b1); add(b2);

   Checkbox ch1 = new Checkbox("one",null,true);
   Checkbox ch2 = new Checkbox("two");
   Checkbox ch3 = new Checkbox("three");
   add(ch1); add(ch2);

   Label l1 = new Label("Hello Egoli");
   Label l2 = new Label("Good Night");
   add(l1); add(l2);

   TextField t = new TextField(20);
   add(t);
   } // end default constructor GUI

   public static void main(String[] args)
   {
   GUI gui = new GUI();
   gui.setVisible(true);
   }
}
```

```
// GUI1.java

import java.awt.*;
import java.awt.event.*;

class GUI1 extends Frame
{
  public GUI1() // default constructor
  {
  setSize(200,200);
  addWindowListener(new WindowAdapter()
  { public void windowClosing(WindowEvent event)
  { System.exit(0); }});

  setLayout(new FlowLayout());

  String[] names = { "Nela", "Alda", "Helen" };

  Choice choice = new Choice();
  choice.addItem(names[0]);
  choice.addItem(names[1]);
  choice.addItem(names[2]);

  add(choice);

  List lst = new List(3,false);
  lst.add("Nela");
  lst.add("Alda");
  lst.add("Helen");

  add(lst);
  } // end default constructor GUI1

  public static void main(String[] args)
  {
  GUI1 gui = new GUI1();
  gui.setVisible(true);
  }
}
```

6.3 ActionListener and ActionEvent

The interface

```
public abstract ActionListener extends EventListener
```

is the listener interface for receiving action events. The class that is interested in processing an action event implements this interface and the object created with that class is registered with a component using the component's `addActionListener` method. When the action event occurs that object's `actionPerformed` method is invoked. The class `ActionEvent` provides for a semantic event which indicates that a component-defined action occured. This high-level event is generated by a component (such as a button) when the component specific action occurs (such as being pressed). The event is passed to every `ActionListener` object that registered to receive such events using the component's `addActionListener` method. The method

```
void actionPerformed(ActionEvent e)   // class ActionListener
```

is invoked when an action occurs.

```
// ButtonWA.java

import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class ButtonWA extends Applet
{
   Button b1, b2;

   TextField t = new TextField(20); // specified number of columns

   public void init()
   {
   b1 = new Button("Button 1");
   b2 = new Button("Button 2");
   b1.addActionListener(new B1());
   b2.addActionListener(new B2());
   add(b1); add(b2); add(t);
   }  // end init

   // The listener interface for receiving action events
   class B1 implements ActionListener
   {
   public void actionPerformed(ActionEvent e)
```

```
   {
   t.setText("Button 1");
   }
   }  // end class B1

   class B2 implements ActionListener
   {
   public void actionPerformed(ActionEvent e)
   {
   t.setText("Button 2");
   }
   }  // end class B2

   // to close the application
   static class WL extends WindowAdapter
   {
   public void windowClosing(WindowEvent e)
   {
   System.exit(0);
   }
   }  // end class WL

   // A main() for application
   public static void main(String[] args)
   {
   ButtonWA applet = new ButtonWA();
   Frame aFrame = new Frame("ButtonWA");
   aFrame.addWindowListener(new WL());
   aFrame.add(applet,BorderLayout.CENTER);
   aFrame.setSize(300,200);

   applet.init();
   applet.start();
   aFrame.setVisible(true);
   }
}
```

6.4 Class Panel

The class `Panel` is a container class in which components can be placed. `Panel` is derived from class `Container`. Components are placed on containers with container method `add`. In Java we can divide a top-level window into panels. Panels act as (smaller) containers for interface elements and can themselves be arranged inside the window. For example we can have one panel for the buttons and another for the text fields. Push button are created with the `Button` class. The class `GridLayout` arranges the components into rows and columns.

```
// ButtonGame.java

import java.awt.*;
import java.awt.event.*;

class ButtonGame extends Frame implements ActionListener
{
  private int nRows, nCols, nButtons;
  private int blankCol, blankRow, clickedRow, clickedCol;
  private Button[][] buttons;
  private TextField textField;

  public ButtonGame()
  {
  setSize(200,200);
  addWindowListener(new WindowAdapter()
  { public void windowClosing(WindowEvent event)
  { System.exit(0); }});

  nRows = 4; nCols = 4; nButtons = nRows*nCols;

  Panel panel = new Panel();
  panel.setLayout(new GridLayout(nRows,nCols));

  buttons = new Button[nRows][nCols];

  for(int nRow=0;nRow<nRows;nRow++)
  {
    for(int nCol=0;nCol<nCols;nCol++)
    {
    buttons[nRow][nCol] = new Button("");
    buttons[nRow][nCol].addActionListener(this);
    panel.add(buttons[nRow][nCol]);
    }
  }
```

```
add("Center",panel);
textField = new TextField("",80);
textField.setEditable(false);
add("South",textField);

int labelsUsed[] = new int[nButtons];

for(int i=0;i<nButtons;i++)
{
  boolean labelUsed;
  int label;
  do
  {
  label = random(nButtons)+1;
  labelUsed = false;
  for(int j=0;j<i;j++)
    labelUsed = ((labelUsed) || (label==labelsUsed[j]));
  } while (labelUsed);

  labelsUsed[i] = label;
  int nRow = i/nCols;
  int nCol = i - nRow*nCols;
  buttons[nRow][nCol].setLabel((new Integer(label)).toString());
  }

  getButtonPosition((new Integer(nButtons)).toString());
  blankRow = clickedRow; blankCol = clickedCol;
  Button blank = buttons[clickedRow][clickedCol];
  blank.setLabel("");    blank.setBackground(Color.green);
} // end constructor ButtonGame

private int random(int k) {return (int)(k*Math.random()-0.1);}

private void getButtonPosition(String label)
{
  for(int nr=0;nr<nRows;nr++)
  {
    for(int nc=0;nc<nCols;nc++)
    {
      if(buttons[nr][nc].getLabel().equals(label))
      {
      clickedRow = nr;  clickedCol = nc;
      textField.setText("["+ nr +',' + nc + "]" + label);
      }
```

```
      }
    }
  }

  public void actionPerformed(ActionEvent e)
  {
    getButtonPosition(e.getActionCommand());
    textField.setText("["+blankRow+","+blankCol+"] => ["
                      +clickedRow +","+clickedCol+"]");
    if(clickedRow == blankRow)
      {
        if(Math.abs(clickedCol-blankCol) == 1)
          moveBlank(blankRow,blankCol,clickedRow,clickedCol);
      }
    else if(clickedCol == blankCol)
      {
      if(Math.abs(clickedRow-blankRow) == 1)
      moveBlank(blankRow,blankCol,clickedRow,clickedCol);
      }
  }

  public void moveBlank(int oldRow,int oldCol,int newRow,int newCol)
  {
  Button oldBlank = buttons[oldRow][oldCol];
  Button newBlank = buttons[newRow][newCol];

  String label = newBlank.getLabel();
  newBlank.setLabel("");    newBlank.setBackground(Color.green);
  oldBlank.setLabel(label); oldBlank.setBackground(Color.lightGray);
  blankRow = newRow;  blankCol = newCol;
  }

  public static void main(String[] args)
  {
  new ButtonGame().setVisible(true);
  }
}
```

In many applications it is useful to group components into separate containers and inserting these containers into the window. For example, for a CAD application we would like to have a drawing area and some button on either side of the drawing area. In such a case it is useful to generate two panels of buttons and insert them in the West and East of the frame window using border layout for the frame window and inserting the drawing canvas into the centre of the frame.

```
// ColorDraw.java

import java.util.*;
import java.awt.*;
import java.awt.event.*;

public class ColorDraw extends Frame
{
   private DrawCanvas drawCanvas;
   private Panel colorButtons;

   public ColorDraw()
   {
   setTitle("Mouse Drawing Application");
   addWindowListener(new WindowAdapter()
   { public void windowClosing(WindowEvent event)
   { System.exit(0); }});

   drawCanvas = new DrawCanvas();
   drawCanvas.setBackground(Color.white);

   int nGradingsInEachColor = 4;
   int nColors =
    nGradingsInEachColor*nGradingsInEachColor*nGradingsInEachColor;

   Panel colorButtons = new Panel();
   int nRows = 16;
   if(nColors < nRows)
   nRows = nColors;
   colorButtons.setLayout(new GridLayout(nRows,0));

   int gradeInterval = 256/(nGradingsInEachColor-1);

   for(int gradeRed=0;gradeRed < nGradingsInEachColor;gradeRed++)
   {
   int red = gradeRed*gradeInterval;
   for(int gradeGreen=0;gradeGreen < nGradingsInEachColor;gradeGreen++)
   {
```

```
   int green = gradeGreen*gradeInterval;
   for(int gradeBlue=0;gradeBlue < nGradingsInEachColor;gradeBlue++)
   {
   int blue = gradeBlue*gradeInterval;

   Button b = new Button("");
   Color color = new Color(red,green,blue);
   b.setBackground(color);
   colorButtons.add(b);
   ColorListener colorListener = new ColorListener(color,drawCanvas);
   b.addActionListener(colorListener);
   }
   }
   } // end outer foor loop

   add("West",colorButtons);
   add("Center",drawCanvas);
   }  // end constructor ColorDraw()

   public static void main(String[] args)
   {
   Frame frame = new ColorDraw();
   frame.setSize(400,300);
   frame.setVisible(true);
   }
}  // end class ColorDraw

class Line
{
   public Point start, end;
   public Color color;

   public Line(Point pStart,Point pEnd,Color color)
   {
   start = new Point(pStart.x,pStart.y);
   end = new Point(pEnd.x,pEnd.y);
   this.color = color;
   }
}  // end class Line

class DrawCanvas extends Canvas
                 implements MouseListener, MouseMotionListener
{
   private Point lineStart = new Point(0,0);
   private Graphics g = getGraphics();
```

```
private Color penColor = Color.blue;
private Vector lines = new Vector();

public DrawCanvas()
{
addMouseListener(this);
addMouseMotionListener(this);
}

public void paint(Graphics g)
{
for(Enumeration enum=lines.elements(); enum.hasMoreElements();)
{
Line line = (Line) enum.nextElement();
g.setColor(line.color);
g.drawLine(line.start.x,line.start.y,line.end.x,line.end.y);
}
g.setColor(penColor);
} // end paint

public void setPenColor(Color color)
{
penColor = color;
} // end setPenColor

public void mousePressed(MouseEvent mouseEvent)
{
if((mouseEvent.getModifiers() & InputEvent.BUTTON1_MASK) ==
    InputEvent.BUTTON1_MASK)
lineStart.move(mouseEvent.getX(),mouseEvent.getY());

if((mouseEvent.getModifiers() & InputEvent.BUTTON3_MASK) ==
    InputEvent.BUTTON3_MASK)
repaint();
} // end mousePressed

public void mouseReleased(MouseEvent mouseEvent) { }
public void mouseEntered(MouseEvent mouseEvent) { }
public void mouseExited(MouseEvent mouseEvent) { }
public void mouseClicked(MouseEvent mouseEvent) { }
public void mouseMoved(MouseEvent mouseEvent) { }

public void mouseDragged(MouseEvent mouseEvent)
{
int newX = mouseEvent.getX();
```

```
   int newY = mouseEvent.getY();
   g = getGraphics();

   g.setColor(penColor);
   g.drawLine(lineStart.x,lineStart.y,newX,newY);
   lines.addElement(new Line(lineStart,new Point(newX,newY),penColor));
   lineStart.move(newX,newY);
   }  // end mouseDragged
}  // end class DrawCanvas

class ColorListener implements ActionListener
{
   private Color color;
   private DrawCanvas destination;

   public ColorListener(Color color,DrawCanvas destination)
   {
   this.color = color;
   this.destination = destination;
   }

   public void actionPerformed(ActionEvent actionEvent)
   {
   destination.setPenColor(color);
   }
}  // end class ColorListener
```

6.5 Mouse Listener and Mouse Event

When we click the mouse of move it a series of events occurs. Java provides some classes which allow to detect that an event has occured and to take some action in response. We can choose which events we want to listen for and which we want to ignore. This event is used both for mouse events (click, enter, exit) and mouse motion events (moves and drags). A `MouseEvent` object is passed to every `MouseListener` or `MouseAdapter` object which registered to receive the interesting mouse events using the component's `addMouseListener()` method. The method

```
Point getPoint()
```

in class `MouseEvent` returns the (x, y) position of the event relative to the source component. The method

```
int getClickCount()
```

returns the number of mouse clicks associated with this event. The methods

```
isAltDown(), isControlDown(), isMetaDown(), isShiftDown()
```

are inherited from class `InputEvent`.

```
// Mouse.java

import java.applet.Applet;
import java.awt.Graphics;
import java.awt.Color;
import java.awt.*;
import java.awt.event.*;

public class Mouse extends Applet
       implements MouseListener, MouseMotionListener
{
  private static final int r = 7;
  private String[] message;
  private int index;
  private int posX = 0;
  private int posY = 0;

  private Point clickPoint = null;
  private MouseEvent mouseEvent = null;

  private boolean bMoved = false;
  private int intMoved = 0;
```

```
private boolean bDragged = false;
private int intDragged = 0;

private String whichButton;

public void init()
{
addMouseListener(this);
addMouseMotionListener(this);
InitButtonTrail();
showStatus("Started Mouse");
}

public void paint(Graphics g)
{
setBackground(Color.white);
if(clickPoint != null)
g.fillOval(clickPoint.x-r,clickPoint.y-r,r*2,r*2);

posX = 150;
posY = 0;

if(mouseEvent != null)
{
g.drawString("Mouse Position: "+ mouseEvent.getPoint(),posX,posY+=15);
g.drawString("Mouse Clicks: "+ mouseEvent.getClickCount(),posX,posY+=15);
g.drawString( "Mouse Button: "+ whichButton,posX,posY+=15);
}

if(bMoved)
{
g.drawString("mouseMoved(): " + spin("mouseMoved"),posX,posY+=15);
bMoved = false;
}

if(bDragged)
{
g.drawString("mouseDragged(): " + spin("mouseDragged"),posX,posY+=15);
bDragged = false;
}

posX = 5;
posY = 0;
int x;
for(x=0;x<message.length;x++)
```

```
    if(message[x] == null) break;
    else g.drawString(message[x],posX,posY+=15);
} // end paint()

private char spin(String mode)
{
int pos = 0;
char c = ' ';

pos = (mode == "mouseMoved" ? intMoved : intDragged);
pos++;

switch(pos) {
  case 1:
    c = '|'; break;
  case 2:
    c = '/'; break;
  case 3:
    c = '-'; break;
  case 4:
    c = '\\'; break;
  default:
    c = '|';
    pos = 1; break;
  }

  if(mode == "mouseMoved") intMoved = pos;
  else intDragged = pos;

return c;
} // end spin

public void mousePressed(MouseEvent event)
{
mouseEvent = event;
whichButton = WhichButton();
InitButtonTrail();
ButtonTrail("mousePressed");
clickPoint = event.getPoint();
repaint();
} // end mousePressed

public void mouseClicked(MouseEvent event)
{
mouseEvent = event;
```

```
whichButton = WhichButton();
ButtonTrail("mouseClicked");
repaint();
} // end mouseClicked

public void mouseReleased(MouseEvent event)
{
mouseEvent = event;
whichButton = WhichButton();
ButtonTrail("mouseReleased");
repaint();
} // end mouseReleased

public void mouseEntered(MouseEvent event)
{
mouseEvent = event;
whichButton = "";
ButtonTrail("mouseEntered");
repaint();
} // end mouseReleased

public void mouseExited(MouseEvent event)
{
mouseEvent = event;
whichButton = "";
ButtonTrail("mouseExited");
repaint();
} // end mouseExited

public void mouseDragged(MouseEvent event)
{
mouseEvent = event;
whichButton = WhichButton();
clickPoint = event.getPoint();
bDragged = true;
repaint();
} // end mouseDragged

public void mouseMoved(MouseEvent event)
{
mouseEvent = event;
whichButton = "";
bMoved = true;
repaint();
} // end mouseMoved
```

```
  private String WhichButton()
  {
  String str = "";
  if(mouseEvent.isShiftDown())
  str = (mouseEvent.isMetaDown() ? "Shift + (Right)" :
                                        "Shift + (Left)");
   else if(mouseEvent.isControlDown())
     str = (mouseEvent.isMetaDown() ? "Ctrl + (Right)" :
                                      "Ctrl + (Left)");
   else if(mouseEvent.isAltDown())
     str = (mouseEvent.isMetaDown() ? "Alt + (Right)" :
                                      "Alt + (Left) also (center)");
    else if(mouseEvent.isMetaDown())
      str = "(right) also Meta + (Left) ";
    else
      str = "(left)";

    return str;
  } // end WhichButton

  void ButtonTrail(String msg)
  {
    if(index < message.length-1)
      message[index] = msg;
    else
      {
      InitButtonTrail();
      message[index] = msg;
      }
    index++;
  } // end ButtonTrail

  void InitButtonTrail()
  {
  index = 0; message = new String[10];
  }

} // end class Mouse
```

6.6 Class Graphics

The `Graphics` class is the abstract base class for all graphics contexts that allow an application to draw onto components that are realized on various devices. Anytime we want to put text or graphics into a window, we need to override the `paint` method from the `Component` class. The `paint` method

```
void paint(Graphics g)
```

has one parameter of type `Graphics`. It paints this component. The `Graphics` class contains all the drawing methods. The method

```
abstract void drawLine(int x1,int y1,int x2,int y2)
```

draws a line between the coordinates `(x1,y1)` and `(x2,y2)`. The line is drawn below and to the left of the logical coordinates. `x1` - the first point x coordinate, `y1` - the first point y coordinate, `x2` - the second point x coordinate, `y2` - the second point y coordinate. The method

```
abstract void drawRect(int x,int y,int width,int height)
```

draws the outline of the specified rectangle. The left and right edges of the rectangle are at `x` and `x+width`. The top and buttom edges are at `y` and `y+height`. The method

```
abstract void drawOval(int x,int y,int width,int height)
```

draws the outline of an oval. The result is a circle or ellipse that fits within the rectangle specified by the `x`, `y`, `width` and `height` argument. The method

```
abstract void drawArc(int x,int y,int width,int height,
                      int startAngle,int arcAngle)
```

draws the outline of a circular or elliptical arc covering the specified rectangle. The method

```
abstract void drawPolygon(int[] xPoint,int[] yPoint,int nPoints)
```

draws a closed polygon defined by the arrays `xPoint` and `yPoint`. Each pair of `(xPoint[i],yPoint[i])` coordinates defines a point. The lines of the polygon can intersect.

The method

```
abstract void dispose()
```

disposes of this graphics context and releases any system resources that it is using. A graphics object cannot be used after dispose has been called. The method

`Graphics getGraphics()`

is a method in class `Component` an creates a graphics context for this component. This method will return `null` if this component is currently not on the screen.

The following program shows the use of the methods `drawLine` and `drawRect`.

```
// GraphA.java

import java.awt.Graphics;

public class GraphA extends java.applet.Applet
{
   double f(double x)
   {
   return (Math.cos(x/5.0)+Math.sin(x/7.0)+2.0)*getSize().height/4.0;
   }

   public void paint(Graphics g)
   {
   for(int x=0;x<getSize().width;x++)
   {
   g.drawLine(x,(int)f(x),x+1,(int)f(x+1));
   }
   g.drawRect(20,20,400,400);
   g.dispose();
   g.drawRect(80,80,200,200);
   }
}
```

The second and third pieces of the applet tag indicate the `width` and the `height` of the applet in pixel. The upper left corner of the applet is always at x-coordinate 0 and y-coordinate 0. The values for `height` and `width` are provided in the HTML file.

```
<HTML>
<COMMENT> HTML file GraphA.html for GraphA.java </COMMENT>
<TITLE> Graph of Mathematical Function </TITLE>
<APPLET CODE="GraphA.class" width=300 height=120>
</APPLET>
</HTML>
```

The following program shows the use of the method `drawArc`. We use this method twice to draw a circle, where half of the circle is filled with color gold and the other half with the color blue.

```
// Arc.java

import java.awt.*;
import java.awt.Frame;
import java.awt.event.*;
import java.awt.Graphics;

public class Arc extends Frame
{
  public Arc()
  {
  setTitle("Arc");
  setSize(400,400);
  addWindowListener(new WindowAdapter()
  { public void windowClosing(WindowEvent event)
  { System.exit(0); }});
  }

  public void paint(Graphics g)
  {
  Color color1 = new Color(255,215,0);
  g.setColor(color1);
  g.drawArc(140,150,150,150,0,180);
  g.fillArc(140,150,150,150,0,-180);

  Color color2 = new Color(255,175,175);
  g.drawArc(140,150,150,150,0,-180);
  g.setColor(color2);
  g.fillArc(140,150,150,150,0,-180);
  }

  public static void main(String[] args)
  {
  Frame f = new Arc();
  f.setVisible(true);
  }
}
```

In the following program we find the time evolution of the logistic map

$$u_{t+1} = 4u_t(1 - u_t), \qquad t = 0, 1, 2, \ldots$$

where $u_0 \in [0, 1]$. We display u_t as a function of t

```
// DrawLogistic.java

import java.awt.Frame;
import java.awt.event.*;
import java.awt.Graphics;

public class DrawLogistic extends Frame
{
  public DrawLogistic()
  {
  addWindowListener(new WindowAdapter()
  { public void windowClosing(WindowEvent event) { System.exit(0); }});
  setTitle("Logistic Map");
  setSize(200,100);
  }

  public void paint(Graphics g)
  {
  int t;
  int size = 280;
  double[] u = new double[size];
  u[0] = 1.0/3.0; // initial value
  for(t=0;t<(size-1);t++)
  u[t+1] = 4.0*u[t]*(1.0 - u[t]);
  int[] us = new int[size];
  for(t=0;t<size;t++)  us[t] = (int) (200*u[t]);
  int[] x = new int[size];
  for(t=0;t<size;t++)  x[t] = 20 + t*3;
  int[] y = new int[size];
  for(t=0;t<size;t++)  y[t] = 400;
  for(t=0;t<size;t++)
  g.drawLine(x[t],y[t],x[t],y[t]-us[t]);
  } // end paint

  public static void main(String[] args)
  {
  Frame f = new DrawLogistic();  f.setVisible(true);
  }
}
```

6.7 Graphics2D Class

The class `Graphics2D` can be considered as a better `Graphics` class. The class `Graphics2D` extends abstract class `Graphics` in order to maintain backwards compatibility. We use `Graphics2D` by casting (type conversion) a `Graphics` reference to a `Graphics2D` reference. The `Graphics2D` class is mainly used with three other classes, namely

```
GeneralPath, AffineTransform, RenderingHints
```

The class `GeneralPath` represents a geometric path constructed from straight lines, quadratic and cubic (Bezier) curves. It can contain multiple subpaths. The method

```
void moveTo(float x,float y)
```

adds a point to the path by moving to the specified coordinates. The method

```
void lineTo(float x,float y)
```

adds a point to the path by drawing a straight line.

```
// Path.java

import java.awt.*;
import java.awt.geom.*;
import java.awt.event.*;
import javax.swing.*;

public class Path extends JFrame
{
   public static void main(String[] args)
   {
   Path test = new Path();
   test.addWindowListener(new WindowAdapter()
   {
   public void windowClosing(WindowEvent e)
   { System.exit(0); }
   });

   test.setBounds(0,0,300,200);
   test.setVisible(true);
   }

   public void paint(Graphics g)
   {
```

```
   Graphics2D g2 = (Graphics2D) g;
   g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                       RenderingHints.VALUE_ANTIALIAS_ON);
   g2.setStroke(new BasicStroke(4.0f));

   GeneralPath polygon =
      new GeneralPath(GeneralPath.WIND_EVEN_ODD);
   polygon.moveTo(40,40);
   polygon.lineTo(250,180);
   polygon.quadTo(120,100,50,160);
   polygon.curveTo(20,90,180,160,220,40);
   polygon.closePath();

   g2.draw(polygon);
   }
}
```

The `Shape` interface provides definitions for objects that represent some form of geometric shape. The Java 2D definition of a shape does not require the shape to enclose an area. A `Shape` object may represent an open curve such as a line or parabola. For example, `RectangularShape` is an implementation of this interface with the subclasses `Arc2D`, `Ellipse2D`, `Rectangle2D`, `RoundRectangle2D`. An application is given below.

```
// Path1.java

import java.awt.*;
import java.awt.geom.*;
import java.awt.event.*;
import javax.swing.*;

public class Path1 extends JFrame
{
   public static void main(String[] args)
   {
   Path1 test = new Path1();
   test.addWindowListener(new WindowAdapter()
   {
   public void windowClosing(WindowEvent e)
   { System.exit(0); }
   });

   test.setBounds(0,0,300,200);
   test.setVisible(true);
   }
```

```
   public void paint(Graphics g)
   {
   Graphics2D g2 = (Graphics2D) g;
   g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                       RenderingHints.VALUE_ANTIALIAS_ON);
   g2.setStroke(new BasicStroke(1.0f));

   Shape round = new GeneralPath(GeneralPath.WIND_EVEN_ODD);
   round = new Ellipse2D.Float(100.0f, 100.0f, 100.0f, 100.0f);
   g2.draw(round);

   Shape square = new GeneralPath(GeneralPath.WIND_EVEN_ODD);
   square = new Rectangle2D.Float(50.0f, 50.0f, 100.0f, 100.0f);
   g2.draw(square);

   Shape rrect = new GeneralPath(GeneralPath.WIND_EVEN_ODD);
   rrect = new RoundRectangle2D.Float(20.0f,60.0f,160.0f,80.0f,20.0f,20.0f);
   g2.draw(rrect);
   }
}
```

The class `AffineTransform` represents a 2D affine transformation which performs a linear mapping in the two-dimensional space. It preserves the straightness and parallelness of lines. The transformation is represented by two dimensional arrays (matrices) of date type `float` or `double`. The method

```
void setToRotation(double theta)
```

sets this transform to a rotation transformation. The method

```
void setToTranslation(double tx,double ty)
```

sets this transform to a translation transformation.

```
// Affine.java

import java.awt.geom.AffineTransform;
import java.util.*;

public class Affine
{
   public static void main(String[] args)
   {
   AffineTransform at = new AffineTransform();
```

```
   double[] src = { 2.1, 4.2, 2.1, 2.6 };
   float[]  dst = new float[4];

   at.rotate(Math.PI);
   // transforms an array of double precision coordinates
   // by this transform (rotate)
   // and stores the result in to an array of floats.
   // Each point is stored as a pair of x, y coordinates.
   // srcOff(set) = 0   dstOff(set) = 0
   // numPts = 2 number of point objects are to be transformed
   at.transform(src,0,dst,0,2);
   System.out.print("source points: ");
   for(int i=0; i<4; i++)
   if((i%2) == 0)
   System.out.print("(" + src[i] + "," + src[i+1] + ")");
   System.out.println("");

   System.out.print("destination points: ");
   for(int i=0; i<4; i++)
   if((i%2) == 0)
   System.out.print("(" + dst[i] + "," + dst[i+1] + ")");
   System.out.println("");
   }
}
// source points:      (2.1,4.2) (2.1,2.6)
// destination points: (-2.1,-4.2) (-2.1,-2.6)
```

The class `RenderingHints` contains rendering hints that can be used by the class `Graphics2D`.

In the following program all four classes are used

`Graphics2D, GeneralPath, AffineTransform, RenderingHints .`

```
// Graph2D2.java

import java.awt.Frame;
import java.awt.event.*;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Color;
import java.awt.geom.AffineTransform;
import java.awt.geom.GeneralPath;
import java.awt.Font;
import java.awt.RenderingHints;
```

```
public class Graph2D2 extends Frame
{
  public Graph2D2()
  {
  addWindowListener(new WindowAdapter()
  { public void windowClosing(WindowEvent event)
  { System.exit(0); }});
  setTitle("Graphics2D Application");
  setSize(400,400);
  }

  public void paint(Graphics g)
  {
  Graphics2D g2d = (Graphics2D) g;  // type conversion
  // the boundary of the figure will look smoother
  g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                       RenderingHints.VALUE_ANTIALIAS_ON);

  GeneralPath path = new GeneralPath(GeneralPath.WIND_EVEN_ODD);
  path.moveTo(0.0f,0.0f);
  path.lineTo(0.0f,125.0f);
  path.quadTo(100.0f,100.0f,225.0f,125.0f);
  path.curveTo(260.0f,100.0f,130.0f,50.0f,225.0f,0.0f);
  path.closePath();

  AffineTransform at = new AffineTransform();
  at.setToRotation(-Math.PI/8.0);
  g2d.transform(at);
  at.setToTranslation(0.0f,150.0f);
  g2d.transform(at);
  g2d.setColor(Color.red);
  g2d.fill(path);
  Font fo = new Font("TimesRoman",Font.PLAIN,40);
  g2d.setFont(fo);
  g2d.setColor(Color.blue);
  g2d.drawString("ISSC",0.0f,0.0f);
  } // end paint

  public static void main(String[] args)
  {
  Frame f = new Graph2D2();
  f.setVisible(true);
  }
}
```

6.8 Color Class

Objects of the `Color` class are used to specify graphics operations. One way to specify color is the three-component RGB code, where R stands for Red, G for Green and B for blue. For example, the RGB code for the color orange is (255,200,0). The three integer numbers identify how much red, green, and blue is used to form the color. Each number can range from 0 to 255. Thus the RGB code (255,255,0) for orange means that it has as much red as it can, it has about `80%` of the green that it could have, and no blue. The `Color` class defines constants for the 13 special colors shown in the table. We can specify

$$256 \times 256 \times 256 = 16777216$$

different RGB code and therefore 16 777 216 colors. For example, $(255, 215, 0)$ is gold, $(127, 255, 212)$ is aquamarine, and $(160, 32, 240)$ is purple. The built-in standard colors are

```
Object              RGB code

Color.black         (0,0,0)
Color.blue          (0,0,255)
Color.cyan          (0,255,255)
Color.darkGray      (64,64,64)
Color.gray          (128,128,128)
Color.green         (0,255,0)
Color.lightGray     (192,192,192)
Color.magenta       (255,0,255)
Color.orange        (255,200,0)
Color.pink          (255,175,175)
Color.red           (255,0,0)
Color.white         (255,255,255)
Color.yellow        (255,255,0)
```

The constructor

```
Color(int r,int g,int b)
```

creates an opaque sRGB color with the specified red, green and blue values in the range $(0, 255)$. The method

```
void setColor(Color)
```

in class `Graphics` sets this graphics context's current color to the specified color.

In the next program we show an application of the `Color` class. We draw a pink, blue and red polygon. The lines of the "polygon" intersect. Explain why ? Try to fix it so that it is really a polygon. In this case a rectangle.

```
// FrameW.java

import java.awt.*;
import java.awt.Frame;
import java.awt.event.*;
import java.awt.Graphics;

public class FrameW extends Frame
{
   public FrameW()
   {
   setTitle("Polygon");
   setSize(400,400);
   addWindowListener(new WindowAdapter()
   { public void windowClosing(WindowEvent event)
   { System.exit(0); }});
   } // end constructor FrameW

   public void paint(Graphics g)
   {
   int x[] = { 40, 180, 30, 180 };
   int y[] = { 40, 30, 180, 180 };
   g.setColor(Color.blue);   // to draw a blue polygon
   g.drawPolygon(x,y,4);

   int u[] = { 20, 40, 320, 400, 420 };
   int v[] = { 25, 100, 80, 200, 400 };
   g.setColor(Color.pink);   // to draw a pink polygon
   g.drawPolygon(u,v,5);

   int a[] = { 80, 250, 300 };
   int b[] = { 100, 300, 320 };
   g.setColor(Color.red);    // to draw a red polygon
   g.drawPolygon(a,b,3);
   g.fillPolygon(a,b,3);
   }

   public static void main(String[] args)
   {
   Frame f = new FrameW();
   f.setVisible(true);
```

```
   }
}
```

A `Frame` is a top-level window with a title and a border. The class `Frame` inherts the method

```
pblic void setVisible(boolean b)
```

from class `Component`. This method shows or hides this component depending on the value of the parameter `b`.

Another application of the `Color` class is shown in the next program. It also demonstrates the use of the method

```
void setXORMode(Color c1)
```

This method sets the paint mode of this graphics context to alternate between this graphics context's current color and the new specified color.

```
// XOR.java

import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class XOR extends Applet
{
   private Color strawberry = new Color(0xcc,0,0x66);
   private Color chocolate = new Color(0x66,0x33,0);
   private Color vanilla = new Color(0xff,0xff,0x99);
   private Color scoop = new Color(0x33,0x99,0xcc);

   private static final int BORDER = 10;

   public void init()
   {
   setBackground(chocolate);

   addMouseListener(
      new MouseAdapter() {
        public void mousePressed(MouseEvent e) {
            Graphics g = getGraphics();
            g.setColor(chocolate);
            g.setXORMode(scoop);
```

```
            Dimension dim = getSize();
            int diameter = dim.height - (2*BORDER);

            int xStart = (dim.width/2) - (diameter/2);
            int yStart = BORDER;

            g.fillOval(xStart,yStart,diameter,diameter);
            g.setPaintMode();
         }
         }
         );   // end addMouseListener
    }  // end init

    public void paint(Graphics g)
    {
    Dimension dim = getSize();
    int width = dim.width/3;
    int height = dim.height;

    g.setColor(strawberry);
    g.fillRect(0,0,width,height);

    g.setColor(vanilla);
    g.fillRect(dim.width-width,0,width,height);
    }  // end paint
}  // end class XOR
```

6.9 Class Image

The `abstract class Image` is the superclass of all classes that represent graphical images.

The

```
public class MemoryImageSource
```

is an implementation of the `ImageProducer` interface which uses an array to produce pixel values for an `Image`. The method

```
Image createImage(ImageProducer produces)
```

in the `Component` class creates an image from the specified image producer. The interface

```
public abstract interface ImageProducer
```

is an interface for objects which can produce the image data for Images. Each image contains an `ImageProducer` which is used to reconstruct the image whenever it is needed. When a image file has been read into an `Image` object, the `Image` object can be displayed using the `drawImage` methods in the `Graphics` class. The method

```
boolean drawImage(Image img,int x,int y,ImageObserver observer)
```

draws as much of the specified image as is currently available. The image is drawn with its top-left corner at (x, y) in the graphics context's coordinate space. The method

```
boolean drawImage(Image img,int x,int y,int width,int height,
                  ImageObserver oberserver)
```

draws an image inside the specified rectangle of this graphics context's coordinate space and is scaled if necessary, where `width` is the width of the rectangle and `height` is the height of the rectangle. The interface

```
public abstract interface ImageObserver
```

is an asynchronous interface for receiving notifications about `Image` information as the `Image` is constructed.

The next two programs show an application of the `drawImage` methods.

```
// HappyFace.java

import java.applet.*;
import java.awt.*;
import java.awt.image.*;

public class HappyFace extends Applet
{
    protected static final int y = Color.yellow.getRGB();
    protected static final int b = Color.black.getRGB();
    protected static final int w = Color.white.getRGB();

    protected static final int[] imageData = {
       w, w, w, w, y, y, y, y, y, y, y, y, w, w, w, w,
       w, w, w, y, y, y, y, y, y, y, y, y, y, w, w, w,
       w, w, y, y, y, y, y, y, y, y, y, y, y, y, w, w,
       w, y, y, y, b, b, y, y, y, y, b, b, y, y, y, w,
       y, y, y, y, b, b, y, y, y, y, b, b, y, y, y, y,
       y, y, y, y, y, y, y, y, y, y, y, y, y, y, y, y,
       y, y, y, y, y, y, y, y, y, y, y, y, y, y, y, y,
       y, y, y, y, y, y, y, y, y, y, y, y, y, y, y, y,
       y, y, y, y, y, y, y, y, y, y, y, y, y, y, y, y,
       y, y, y, b, y, y, y, y, y, y, y, y, b, y, y, y,
       y, y, y, y, b, y, y, y, y, y, y, b, y, y, y, y,
       y, y, y, y, y, b, b, y, y, b, b, y, y, y, y, y,
       w, y, y, y, y, y, y, b, b, y, y, y, y, y, y, w,
       w, w, y, y, y, y, y, y, y, y, y, y, y, y, w, w,
       w, w, w, y, y, y, y, y, y, y, y, y, y, w, w, w,
       w, w, w, w, y, y, y, y, y, y, y, y, w, w, w, w
       };

    Image happy;

    public void init()
    {
    happy =
     createImage(new MemoryImageSource(16,16,imageData,0,16));
    }

    public void paint(Graphics g)
    {
    g.drawImage(happy,10,10,128,128,this);
    }
}
```

In the next program we display a 400 x 400 image representating a fade from black to blue along the x-axis and a fade from black to red along the y-axis.

```
// Pixel.java

import java.awt.*;
import java.awt.event.*; // includes WindowEvent, WindowListener
import java.awt.Graphics;
import java.awt.image.*;

public class Pixel extends Frame
{
   public Pixel()
   {
   setTitle("400 x 400 Image");
   setSize(400,400);
   addWindowListener(new WindowAdapter()
   { public void windowClosing(WindowEvent event)
   { System.exit(0); }});
   }

   public void paint(Graphics g)
   {
   int w = 400;  int h = 400;
   int pix[] = new int[w*h];
   int index = 0;
   for(int y=0; y < h; y++)
   {
   int red = (y*255)/(h-1);
   for(int x=0; x < h; x++)
   {
   int blue = (x*255)/(w-1);
   pix[index++] = (255 << 24) | (red << 16) | blue;
   }
   }
   Image img;
   img = createImage(new MemoryImageSource(w,h,pix,0,w));
   g.drawImage(img,0,0,null);
   }

   public static void main(String[] args)
   {
   Frame f = new Pixel();  f.setVisible(true);
   }
}
```

The Java image display mechanism has the ability to shrink or stretch images to fit a particular size. Normally, when we display an image, we only specify the x and y coordinates for the upper left corner of the image. We can, however, specify, an alternate width and height for the image. Java automatically scales the images to fit the new width and height. The following applet takes an image and displays it stretched and shrunk.

```
// Media.java

import java.applet.Applet;
import java.awt.Graphics;
import java.awt.Image;
import java.awt.MediaTracker;

public class Media extends Applet
{
   Image image;

   public void init()
   {
   image = getImage(getDocumentBase(),"output.jpg");

   MediaTracker tracker = new MediaTracker(this);
   tracker.addImage(image,0);

   try { tracker.waitForAll(); }
   catch(Exception ignore) {  }
   }

   public void paint(Graphics g)
   {
   int width = image.getWidth(this);
   int height = image.getHeight(this);

   // draw the image in its normal size
   g.drawImage(image,10,10,width,height,this);
   // draw the image at half-size
   g.drawImage(image,width+20,10,width/2,height/2,this);
   // draw the image at twice its size
   g.drawImage(image,width*3/2+30,10,width*2,height*2,this);
   }
}
```

6.10 Class Toolkit

The AWT-Abstract Window Toolkit-uses, in fact, a "toolkit" to do windowing tasks. The `Component` class provides a

```
getToolkit()
```

method to give a reference to the toolkit. The toolkit for the system will be a subclass of the abstract Toolkit class created for our host. Toolkit uses various native peer code to access the system GUI. There is also the static method

```
java.awt.Toolkit.getDefaultToolkit()
```

in the abstract `Toolkit` class that returns a reference to the toolkit.

Toolkit provides all sorts of useful services:

1. Get images for applications
2. Printing the screen
3. Info on screen size and resolution:

```
java.awt.Toolkit.getScreenSize()
java.awt.Toolkit.getScreenResolution()
```

4. Beep the keyboard

```
java.awt.Toolkit.beep()
```

5. System colors (use index constants in `SystemColor` class to relate index to a given window component):

```
java.awt.Toolkit.loadSystemColors(int[] systemColors)
```

For example,to create a frame that is sized to a particular proportion to the screen's size and in particular location is illustrated by the following code:

```
// PlaceFrame.java

import java.awt.*;
import java.awt.event.*;

public class PlaceFrame extends Frame
{
  public static void main(String [] arg)
  {
  PlaceFrame pf = new PlaceFrame();
  pf.setTitle("Centered Frame");

  // need this listener to get the window to close.
  WindowListener listener = new WindowAdapter() {
  public void windowClosing (WindowEvent e){
    System.exit(0);
  }
  };

  pf.addWindowListener(listener);

  // need the toolkit to get info on system.
  Toolkit tk = Toolkit.getDefaultToolkit();

  // get the screen dimensions.
  Dimension screen = tk.getScreenSize();

  // make the frame 1/4 size of screen.
  int fw = (int) (screen.getWidth()/4);
  int fh = (int) (screen.getWidth()/4);
  pf.setSize(fw,fh);

   // and place it in center of screen.
  int lx = (int) (screen.getWidth()*3/8);
  int ly = (int) (screen.getHeight()*3/8);
  pf.setLocation(lx,ly);

  pf.setVisible(true);
  }
}
```

The following program shows that use of the `beep()` method.

```
// Beep.java

import java.util.Timer;
import java.util.TimerTask;
import java.awt.Toolkit;

// Demo that uses java.util.Timer to schedule a task
// to execute once 5 seconds have passed.

public class Beep
{
   Toolkit toolkit;
   Timer timer;

   public Beep(int sec)
   {
   toolkit = Toolkit.getDefaultToolkit();
   timer = new Timer();
   timer.schedule(new Task(),sec*1000);
   }

   class Task extends TimerTask
   {
   public void run()
   {
   System.out.println("Time's up!");
   toolkit.beep();
   System.exit(0);   // stops the AWT thread (and everything else)
   }
   }

   public static void main(String args[])
   {
   System.out.println("About to schedule task.");
   new Beep(5);
   System.out.println("Task scheduled.");
   }
}
```

Chapter 7

Exception Handling

7.1 Introduction

An *exception* is an error that occurs at run time. An example is division by zero. Another example is if we try to open a file which does not exist an error will occur. We can catch this, display an error message and ask for another filename to be specified. When an error of this type occurs an exception is generated and Java provides a set of language constructs which allow us to handle them. Using Java's exception-handling subsystem we can in a structered and controlled manner handle run-time error. Java exception handling like C++ is built upon three keywords

`try, catch, throw`

To respond to an exception, the call to the method that produces it must be placed within a `try` block. Thus the `try` block indicates the code which is to be monitored for the exceptions listed in the `catch` expressions. A `try` block is a block of code beginning with the `try` keyword followed by a left and right curly brace. Every `try` block is associated with one or more `catch` blocks. When any method in the `try` block throws any type of exception, execution of the `try` block ceases. Program control passes immediately to the associated `catch` block. We can use an `catch` expression to indicate what to do when certain classes of exception occur (e.g. resolve the problem or generate a warning message). If no `catch` expression is included, the defining methods must use the `throw` clause.

In C++ we can throw an instance of any class when an expectional situation arises. This is not the case in Java. In Java an exception is always an instance of the class `Throwable`, i.e. it must be directly or indirectly derived from `Throwable`.

7.2 The Exception Class

All exceptions are subclasses of the `Exception` class. There are more than 200 exceptions defined in Java. We can also write our own exceptions if we cannot find an existing one which is suitable for our case. The

```
public class Exception extends Throwable
```

are a form of `Throwable` that indicates conditions that a reasonable application might want to catch. The method

```
public String getMessage()  // in class Throwable
```

returns the detail message of this throwable object. It returns `null` if this `Throwable` does not have a detail message.

Java distinguishes between internal errors caused by a problem in Java run-time system and exceptions which are problems Java encounters when executing the code. For example

```
NegativeArraySizeException,  FileNotFoundException
```

report exceptional situations which might occur when using our classes. Furthermore, these exceptional cases not not indicate a coding error in our class. They would typically occur due to illegal information obtained when our class communicates with the outside world.

For example consider the following Java application

```
// Except.java

public class Except
{
   public static void main(String[] args)
   {
   System.out.println(args[2]);
   }
}
```

After compiling to obtain the `class` file we enter at the command line

```
java Except Good Morning
```

This leads to the error message

```
java.lang.ArrayIndexOutofBoundsException: 2
```

since

```
args[0] -> Good  args[1] -> Morning
```

7.3 Examples

Besides using the built-in exceptions of Java we can also create our own exceptions. In the following program an exception is thrown when the first `int` number is larger than the second `int` number in the method `numbers`. The second call to the method `numbers` throws an exception since the first integer is larger than the second integer.

```
// Except.java

import java.io.*;
import java.lang.Exception;

public class Except
{
   public static void main(String[] args)
   {
   try
   {
   System.out.println(check(60,80)); // method check called
   System.out.println(check(75,15)); // method check called <- problem
   System.out.println(check(90,100));// method check called
   }  // end try block
   catch(Exception e)
   {
   System.out.println("Caught exception: " + e.getMessage());
   }  // end catch block
   }  // end main

   // method numbers
   static boolean check(int first,int second) throws Exception
   {
   boolean returnCode = false;
   if(first > second)
   {
   throw new Exception("invalid values");
   }
   if(first <= second)  { returnCode = true; }

   return returnCode;
   } // end method check
}
```

The output is:

```
true
Caught exception: invalid values
```

To define an exception we can also create a subclass of the `Exception` class or one of its subclasses. We apply this to the `DivideByZeroException` class.

```
// DivideByZeroException.java

public class DivideByZeroException extends Exception
{
public Object offender;

public DivideByZeroException(Object obj)
{
super("Divide by zero in " + obj);
offender = obj;
}
}
```

A program which uses this class is given by

```
// DivideZero.java

public class DivideZero
{
public int x, y;
public DivideZero(int a,int b) { x=a; y=b; }

public static void main(String args[])
{
double x = Math.random();
int i = (int) (4.0*x);
DivideZero temp = new DivideZero(5,i);

try { temp.test(); }
catch(DivideByZeroException e) { System.out.println("Oops"); }
}  // end main

public void test() throws DivideByZeroException
{
if(y == 0) throw new DivideByZeroException(this);
else System.out.println(x/y);
}  // end test

}  // end class DivideZero
```

The following example shows the use of multiple `catch` clauses and of the keyword

`finally` .

The keyword `finally` is used to specify a control block to be executed after all processing of a `try` block, including the processing for any exceptions (regardless of whether those exceptions are handled by a `catch` clause within the `try` block or not). Multiple `catch` clauses are typically used in file input and output operations.

```
// Copy.java

import java.io.*;

public class Copy
{
   public static void main(String[] args)
   {
   if(args.length != 2)
   {
   System.out.println("Usage: java Copy inFile outFile");
   System.exit(0);
   }
   FileInputStream fin = null;
   FileOutputStream fout = null;
   try
   {
   fin = new FileInputStream(args[0]);
   fout = new FileOutputStream(args[1]);

   int aByte = 0;
   do
   {
   aByte = fin.read();
   if(aByte != -1)
   fout.write((byte) aByte);
   }
   while(aByte != -1);

   } // end try block

   // multiple catch clauses
   catch(EOFException e)
   { }
   catch(IOException ex)
   { System.out.println("Error when copying file."); }
```

```
   finally
   {
   try
   {
   fin.close();
   fout.close();
   }
   catch(Exception e)
   { }
   } // end finally
   } // end main
}
```

Chapter 8

File Manipulations

8.1 Introduction

The Java IO stream library allows to perform reading and writing with the console, a file, a block of memory, or even across the Internet. It is possible by inheriting from `InputStream` and `OutputStream` to create new types of input and output objects.

We output data using a `DataOutputStream` that is connected to a `FileOutputStream` via a technique called chaining of stream objects. When the `DataOutputStream` object output is created, its constructor is supplied a `FileOutputStream` object as an argument. The statement creates a `DataOutputStream` object named output associated with the file `filename`. The argument `filename` is passed to the `FileOutputStream` constructor which opens the file. We input data using the `DataInputStream` that is connected to the `FileInputStream`.

`DataInputStream` allows us to read all basic data types as well as `String` objects. All the methods names start with `read` such as

```
readByte(), readShort(), readInt(), readLong(),

readFloat(), readDouble(),

readChar(), readUTF(), readBoolean()
```

The complement to `DataInputStream` is `DataOutputStream` which formats each of the basic data types and `String` objects onto a stream in such a way that any `DataInputStream`, on any machine, can read them. All the methods name start with `write` such as

```
writeByte(), writeShort(), writeInt(), writeLong(),

writeFloat(), writeDouble()
```

`writeChar(), writeUTF(), writeBoolean()`

The methods `writeFloat()` and `writeDouble()` perform binary file manipluation, i.e. the `float` and `double` take 4 and 8 bytes for storage, respectively .

The method

```
public int available() throws IOException
```

is inherited from class `java.io.FilterInputStream`. It returns the number of bytes that can be read from this input stream without blocking.

The method

```
public void flush() throws IOException
```

flushes this data output stream. This forces any buffered ouput bytes to be written out to the stream. The method

```
public void close() throws IOException
```

closes this output stream and releases any system resources with the stream.

Let us look at some of the read methods. The method

```
byte readByte()
```

returns the next byte of this input stream as a signed 8 bit byte. In most applications we type convert the `byte` to `char` after reading the `byte`. The method

```
double readDouble()
```

reads eight input bytes and returns a `double` value. The method

```
String readUTF()
```

reads from the stream in a representation of a Unicode character string encoded in UTF format. This string of characters is then returned as a String.

The method

```
String readLine()
```

in class `DataInputStream` is deprecated. We have to use the method

```
String readLine()
```

in class `BufferedReader` which reads a line of text. A line is considered to be terminated by any one of a line feed (`'\n'`), a carriage return (`'\r'`), or a carriage return followed immediately by a line feed. The return string contains the contents of the line, not including any line termination characters, or `null` if the end of the stream has been reached. The class `BufferedReader` reads from character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays and lines.

Let us now look at some of the write methods. The method

```
void writeByte(int v)
```

writes out a byte to the underlying output stream as 1-byte value. If no exception is thrown the counter `written` is incremented by 1. The method

```
void writeBytes(String s)
```

writes out the string to the underlying output stream as a sequence of bytes. Each character in the string is written out, in sequence, by discarding its high eight bits. If no exception is thrown the counter `written` is incremented by the length of `s`. For writing strings we can also use the method

```
void writeUTF(String str)
```

It writes a string to the underlying output stream using UTF-8 encoding in a machine independent manner. The method

```
void writeInt(int v)
```

writes an `int` to the underlying output stream as four bytes, high byte first. The method

```
void writeDouble(double v)
```

converts the `double` argument to a `long` using the `doubleToLongBits` method in the class `Double`, and then writes that `long` value to the underlying output stream as an 8-byte quantity, high byte first.

8.2 Examples

We show how to write an `int`, `char`, `String`, and `boolean` to a file. Then we read the contents back and display it. The file name is `myout.dat`.

```
// WriteTo.java

import java.io.*;
import java.util.*;

public class WriteTo
{
   public static void main(String[] args) throws IOException
   {
   DataOutputStream output =
   new DataOutputStream(new FileOutputStream("myout.dat"));

   int x = -56;   output.writeInt(x);
   char c = 'X';  output.writeChar(c);
   String s = new String("willi");  output.writeUTF(s);
   boolean b = true;  output.writeBoolean(b);

   output.flush();  output.close();

   int y;
   char d;
   String t;
   boolean bool;

   DataInputStream input =
   new DataInputStream(new FileInputStream("myout.dat"));

   y = input.readInt();
   d = input.readChar();
   t = input.readUTF();
   bool = input.readBoolean();

   System.out.println("y = " + y);
   System.out.println("d = " + d);
   System.out.println("t = " + t);
   System.out.println("bool = " + bool);

   input.close();
   }
}
```

In the following program we include exception handling.

```
// FileExc.java

import java.io.*;
import java.lang.Exception;

public class FileExc
{
   public static void main(String[] args)
   {
   int i = 92345;
   int j = -6789;
   double x = 3.45;
   String name1 = new String("olli");
   String name2 = new String("lipolo");
   int m;
   int n;
   double y;
   String name3;
   String name4;

   DataOutputStream output;

   try
   {
   output = new DataOutputStream(new FileOutputStream("mydata.dat"));

   output.writeInt(i);
   output.writeUTF(name1);
   output.writeDouble(x);
   output.writeUTF(name2);
   output.writeInt(j);

   try
   {
   output.flush();  output.close();
   }
   catch(IOException e)
   {
   System.err.println("File not opened properly\n" + e.toString());
   System.exit(1);
   }
   }
   catch(IOException e)
```

```
   {
   System.err.println("File not opened properly\n" + e.toString());
   System.exit(1);
   }

   try
   {
   DataInputStream in;
   boolean moreRecords = true;

   FileInputStream fin = new FileInputStream("mydata.dat");
   in = new DataInputStream(fin);

   try
   {
   m = in.readInt();
   System.out.println("m = " + m);
   name3 = in.readUTF();
   System.out.println("name3 = " + name3);
   y = in.readDouble();
   System.out.println("y = " + y);
   name4 = in.readUTF();
   System.out.println("name4 = " + name4);
   n = in.readInt();
   System.out.println("n = " + n);
   }
   catch(EOFException eof)
   {
   moreRecords = false;
   }
   catch(IOException e)
   {
   System.err.println("Error during read from file\n" + e.toString());
   System.exit(1);
   }

   }
   catch(IOException e)
   {
   System.err.println("File not opened properly\n" + e.toString());
   System.exit(1);
   }

   }
}
```

In the following program we use a `for` loop to write data of data type `double` to a file. Then we read the data back using a `while` loop. Exception handling is included.

```
// FileMani.java

import java.io.*;
import java.lang.Exception;

public class FileMani
{
   public static void main(String args[])
   {
   DataOutputStream output;

   try
   {
   output = new DataOutputStream(new FileOutputStream("timeev.dat"));

   int T = 10;
   double x0 = 0.618;
   double x1;
   output.writeDouble(x0);
   System.out.println("The output is " + x0);

   int t;

   for(t=0; t<T; t++)
   {
   x1 = 4.0*x0*(1.0 - x0);
   System.out.println("The output is " + x1);
   output.writeDouble(x1);
   x0 = x1;
   }

   try
   {
   output.flush();
   output.close();
   }
   catch(IOException e)
   {
   System.err.println("File not closed properly\n" + e.toString());
   System.exit(1);
   }
```

```
   }

   catch(IOException e)
   {
   System.err.println("File not opened properly\n" + e.toString());
   System.exit(1);
   }

   System.out.println("\nReading file:");

   try
   {
   FileInputStream fin = new FileInputStream("timeev.dat");
   DataInputStream in = new DataInputStream(fin);

   while(true)
   System.out.print(in.readDouble() + " ");
   }
   catch(Exception e) {  }
   }
}
```

In the following three programs we show how the method `readLine()` in class `BufferedReader` is used. In the first program we read in the file as a string using `readLine()` from the class `BufferedReader`. The ASCII file `test.dat` contains

```
1.234 Peter 23 6789012345678
```

The class `StringTokenizer` allows an application to break a string into tokens. The method `String nextToken()` returns the next token from this string tokenizer.

```
// FileIn.java

import java.io.*;
import java.util.*;
import java.math.*;

public class FileIn
{
   public static void main(String[] argv) throws IOException
   {
   FileInputStream fin = new FileInputStream("test.dat");
   BufferedReader in = new BufferedReader(new InputStreamReader(fin));

   String str = in.readLine();

   StringTokenizer tokenizer = new StringTokenizer(str);

   String s = tokenizer.nextToken();

   double x = new Double(s).doubleValue();
   System.out.println("x = " + x);

   s = tokenizer.nextToken();
   System.out.println("s = " + s);

   s = tokenizer.nextToken();
   int i = new Integer(s).intValue();
   System.out.println("i = " + i);

   s = tokenizer.nextToken();
   BigInteger bi = new BigInteger(s);
   System.out.println("bi = " + bi);

   fin.close();
   }
}
```

The following program finds the number of lines in the file `data.dat`. For example if the file `data.dat` contains the lines

```
xxx yyy
123
olloo illi
```

the output will be 3. In Java the `null` keyword is used to identify a variable as not referencing any object. The `null` keyword cannot be assigned to a variable that is declared with a basic data type. It is left as an exercise to include exception handling for the program.

```
// NoOfLines.java

import java.io.*;
import java.util.*;

public class NoOfLines
{
   public static void main(String[] args) throws IOException
   {
   FileInputStream fin = new FileInputStream("data.dat");
   BufferedReader in;
   in = new BufferedReader(new InputStreamReader(fin));

   int count = 0;

   while(in.readLine() != null)
   {
   count++;
   }
   System.out.println("count = " + count);

   } // end main
}
```

The following program counts the number of occurrences of the name `Miller` in the file `data.dat`. All the lines in the file `data.dat` are read using the command

```
while(!(null == (buffer = in.readLine())))
```

The `StringTokenizer` class is used.

```
// NoOfNames.java

import java.io.*;
import java.util.*;

public class NoOfNames
{
   public static void main(String[] args) throws IOException
   {
   FileInputStream fin = new FileInputStream("data.dat");
   BufferedReader in;
   in = new BufferedReader(new InputStreamReader(fin));

   int count = 0;
   String buffer;

   String name = new String("Miller");

   while(!(null == (buffer = in.readLine())))
   {
   StringTokenizer tok = new StringTokenizer(buffer);

   while(tok.hasMoreTokens())
   {
   String s = tok.nextToken();
   if(s.equals(name))
   count++;
   }
   }
   System.out.println("Name " + name + " occurse " + count + " times");
   }
}
```

In the following three programs we show how `readByte` is used in file manipulation. A file can be considered as a sequence of bytes.

In the following program we read in the file character by character or more precisely byte by byte. Then we do type conversion to `char`. Everytime we read a byte we test for the end of the file. This is done with the method `available()`. The program counts the number of characters in the file.

```
// EndOfFile.java

import java.io.*;

public class EndOfFile
{
   public static void main(String[] args)
   {
   long count = 0L;

   try
   {
   DataInputStream in = new DataInputStream(
                         new BufferedInputStream(
                          new FileInputStream("EndOfFile.java")));

   while(in.available() != 0)
   {
   System.out.print((char) in.readByte());  // type conversion
   count++;
   }
   in.close();
   }  // end try block

   catch(IOException e)
   {
   System.err.println("IOException");
   }

   System.out.println("count = " + count);
   }
}
```

In the following program we find the number of curly brackets to the left { and number of curly brackets to the right } in a file. This program can be used to check for Java, C and C++ programs whether the number of left and right curly brackets match. We recall that the curly bracket to the left { indicates a block begin and a curly bracket to the right } indicates a block end.

```
// Oops.java

import java.io.*;

public class Oops
{
   public static void main(String[] args)
   {
   char cl = '{';
   int countcl = 0;
   char cr = '}';
   int countcr = 0;

   try
   {
   DataInputStream in = new DataInputStream(
                          new BufferedInputStream(
                           new FileInputStream("Oops.java")));

   while(in.available() != 0)
   {
   char c = (char) in.readByte();

   if(c == cl)  countcl++;
   if(c == cr)  countcr++;
   }
   }  // end try block

   catch(IOException e)
   { System.err.println("IOException"); }

   System.out.println("countcl = " + countcl);
   System.out.println("countcr = " + countcr);
   }
}
```

In the following program we find the *checksum* (modulo 65535) of a file using the ASCII table. Again we read the data input stream byte by byte.

```
// ReadFile.java

import java.io.*;

public class ReadFile
{
   public static void main(String[] args)
   {
   long count = 0L;
   int checksum = 0;
   char c;

    try
    {
    DataInputStream in =
        new DataInputStream(
          new BufferedInputStream(
            new FileInputStream("ReadFile.java")));
    while(in.available() != 0)
    {
    c = (char) in.readByte();
    checksum += (int) c;       // type conversion ASCII table
    while(checksum > 65535)
    {
    checksum -= 65535;
    }
    System.out.print(c);
    count++;
    }
    }

    catch(IOException e)
    {
    System.err.println("Error readin file");
    }

    System.out.println("count = " + count);
    System.out.println("checksum = " + checksum);
    }
}
```

In the following program we copy a file byte for byte.

```
// Copy.java

import java.io.*;

class Copy
{
   static public void main(String[] args)
   {
   if(args.length != 2)
   {
   System.out.println("Usage: java Copy inputfile outputfile");
   System.exit(0);
   }
   FileInputStream fin = null;
   FileOutputStream fout = null;
   try
   {
   fin = new FileInputStream(args[0]);
   fout = new FileOutputStream(args[1]);
   int inputByte = fin.read();
   while(inputByte != -1)
   {
   fout.write((byte) inputByte);
   inputByte = fin.read();
   }
   } // end try
   catch(EOFException eofError) {          }
   catch(FileNotFoundException notFoundError)
   { System.err.println("File " + args[0] + " not found!"); }
   catch(IOException ioError)
   { System.err.println(ioError.getMessage()); }

   finally
   {
   try
   {
   if(fin != null) fin.close();
   if(fout != null) fout.close();
   }
   catch(IOException error) {  }
   } // end finally
   } // end main
}
```

8.3 FileReader and FileWriter

The `Reader` class is an abstract class which has a set of subclasses for reading text. We consider the two subclasses `FileReader` and `BufferedReader`. The constructor `FileReader(file)` in the class `FileReader` creates a connection between the `file` specified and the application. The class `BufferedReader` was already described above. The class `Writer` is an abstract class which has a set of subclasses for writing character files. We consider two subclasses `FileWriter` and `BufferedWriter`. The class `FileWriter` is a class for writing character files. The constructors of this class assume that the default character encoding and the default byte-buffer size are acceptable. The class `BufferedWriter` writes text to a character-output stream, buffering characters so as to provide for the efficient writing of single character arrays, and strings. Why is the size of the file `out1.dat` 11 bytes?

```
// WriteLine.java

import java.io.*;

public class WriteLine
{
   public static void main(String[] argv) throws IOException
   {
   BufferedWriter wr =
       new BufferedWriter(new FileWriter("out1.dat"));
   String s1 = new String("willi");
   wr.write(s1,0,s1.length());
   wr.newLine();
   String s2 = new String("hans");
   wr.write(s2,0,s2.length());
   wr.flush();  wr.close();

   DataOutputStream dos =
       new DataOutputStream(new FileOutputStream("out2.dat"));
   String t1 = new String("willi");
   dos.writeUTF(t1);
   dos.flush();  dos.close();
   }
}
```

A *checksum* is a count of the number of bits in a transmission unit that is included with the unit so that the receiver can check to see whether the same number of bits arrived. The class `CRC32` can be used to calculate the CRC-32 of a data stream. The method

```
void update(byte[] b)
```

updates checksum with specified bytes. In the `String` class we have the method

`byte[] getBytes()`

which converts this `String` into bytes according to the platform's default character encoding, storing the result into a a new byte array. The method

`public long getValue()`

is in class `CRC32` and returns a CRC-32 value.

In the following two programs we show how the classes `FileReader`, `BufferedReader`, `FileWriter`, `BufferedWriter` and `CRC32` can be used.

```
// WriteSum.java

import java.io.*;
import java.util.zip.CRC32;

public class WriteSum
{
   static final String[] stringlist =
    { "International", "School", "for", "Scientific", "Computing" };

   public static void main(String[] args) throws IOException
   {
   FileWriter fw = new FileWriter("out.dat");
   BufferedWriter bw = new BufferedWriter(fw);
   CRC32 sum = new CRC32();

   bw.write(Integer.toString(stringlist.length));
   bw.newLine();

   int i = 0;
   while(i < stringlist.length)
   {
   String name = stringlist[i];
   bw.write(name);
   bw.newLine();
   sum.update(name.getBytes());
   i++;
   }

   bw.write(Long.toString(sum.getValue()));
   bw.newLine();
   bw.close();
   } // end main
}
```

```
// ReadSum.java

import java.io.*;
import java.util.zip.CRC32;

public class ReadSum
{
   public static void main(String[] args) throws IOException
   {
   FileReader fr = new FileReader("out.dat");
   BufferedReader br = new BufferedReader(fr);
   CRC32 sum = new CRC32();

   int len = Integer.parseInt(br.readLine());

   String stringlist[] = new String[len];

   int i = 0;
   while(i < len)
   {
   stringlist[i] = br.readLine();
   sum.update(stringlist[i].getBytes());
   i++;
   }

   long cs = Long.parseLong(br.readLine());

   br.close();

   if(cs != sum.getValue())
   {
   System.err.println("bad checksum");
   }
   else
   {
   for(i=0; i < len; i++)
   {
   System.out.println(stringlist[i]);
   }
   }
   } // end main
}
```

8.4 File Class

The class

```
public class File extends Object
implements Serializable, Comparable
```

is an abstract representation of file and directory pathnames. User interfaces and operating systems use system-dependent pathname strings to name files and directories. This class presents an abstract, system-independent view of hierarchical pathnames. An abstract pathname has two components. First an optional system-dependent prefix string, such as a disk-drive specifier, `"/"` for the UNIX root directory, or `"\\"` for a Win32 UNC pathname, and second a sequence of zero or more non-empty string names.

The `File` class can be used to open an disc file and write and read data to the file. It also provides information about the file. There are three constructors in the class. In the constructor

```
File(String)
```

the `String` is a file path name which is converted to an abstract path name. In the constructor

```
File(File parent,String child)
```

the parent is a directory which contains the child file. In the constructor

```
File(String parent,String child)
```

the parent directory is specified as a `String` and contains the child file.

The class contains a method

```
boolean exists()
```

which we can use to check whether a file exists or not. Additionally there is a large set of methods for finding out details about the file. These methods are

```
boolean canRead()
boolean canWrite()
boolean exists()
File getAbsoluteFile()
String getAbsolutePath()
String getName()
String getParent()
String getPath()
boolean isDirectory()
boolean isFile()
long length()
```

Furthermore the class has the methods `boolean mkdir()` to create the directory named by this abstract pathname and `boolean delete()` which deletes the file or directory denoted by this abstract pathname.

If we run the program under Linux we have to modify the lines

```
File f = new File("c:\\books\\java\\data.dat");
```

and

```
File newf = new File("c:\\books\\java\\mydata");
```

The Linux operating system is made up of several directories and many different files. The `/dev` directory contains special files known as device files. These files are used to access all of the different types of hardware on our Linux system.

```
// FileClass.java

import java.io.*;

public class FileClass
{
   public FileClass() throws IOException  // constructor
   {
   boolean tempFile = false;
   boolean deleted = false;

   File f = new File("c:\\books\\java\\data.dat");

   if(!f.exists())
   {
   tempFile = f.createNewFile();
   System.out.println("try to create temp file");
   }
   if(f.exists()) printFileDetails(f);
   else
   System.out.println("File does not exist and cannot be created");
   if(tempFile)
   {
   deleted = f.delete();
   if(deleted) System.out.println("temp file deleted");
   else System.out.println("temp file not deleted");
   }
   }  // end constructor FileClass

   private void printFileDetails(File f)
```

```
   {
   System.out.println("File name: " + f.getName());
   System.out.println("Absolute File: " + f.getAbsoluteFile());
   System.out.println("Path: " + f.getAbsolutePath());
   System.out.println("File length: " + f.length());

   if(f.isFile()) System.out.println("File is normal");
   if(f.isHidden()) System.out.println("File is not hidden");
   if(f.isDirectory()) System.out.println("File is a directory");
   if(f.canRead()) System.out.println("File is readable");
   else System.out.println("File is not readable");
   if(f.canWrite()) System.out.println("File is writeable");
   else System.out.println("File is not writeable");

   System.out.println("Path separator is: " + f.pathSeparator);
   System.out.println("Name separator is: " + f.separator);
   } // end method printFileDetails


   public static void main(String[] args) throws IOException
   {
   new FileClass();

   File newf = new File("c:\\books\\java\\mydata");
   boolean b = newf.mkdir();
   System.out.println("b = " + b);
   }
}
```

To obtain the properties of a file or directory one uses Java's `File` class. For example, the following code displays the directory tree of the file system. Recursion is used for the method `subTree()`. After compiling the file `DirTree` we run the program with

```
java DirTree ..\..
```

```
// DirTree.java

import java.io.*;

class DirTree
{
   static int nDir, nFiles; // number of directories, files

   static public void main(String[] args)
   {
   if(args.length != 1)
   {
   System.err.println("Usage: java DirTree <starting dir>");
   }

   File root = new File(args[0]);
   nDir = 0;
   nFiles = 0;

   if(!root.exists())
   {
   System.err.println("directory " + root + " not found");
   System.exit(0);
   }

   subTree(root,"");
   System.out.println(nDir + " directories with " + nFiles + " files");
   } // end main

   static private void subTree(File root,String prefix)
   {
   System.out.println(prefix + root.getName());
   prefix = "+ " + prefix;

   String[] dirList = root.list();

   for(int entryNo = 0; entryNo < dirList.length; entryNo++)
```

```
   {
   String entryName = root.getAbsolutePath() + File.separatorChar
                    + dirList[entryNo];

   File item = new File(entryName);
   if(item.isDirectory())
   {
   ++nDir;
   subTree(item,prefix);
   }
   else
   ++nFiles;
   }
   } // end subTree
}
```

Finally we show how the `File` class can be used to copy a file.

```
// Copy1.java

import java.io.*;

public class Copy1
{
   public static void main(String[] args)
   {
   if(args.length != 2)
   {
   System.out.println("usage: java Copy1 source dest");
   return;
   }

   File src_f = new File(args[0]);
   File dst_f = new File(args[1]);
   if(!src_f.exists() || !src_f.canRead())
   {
   System.out.println("cannot find source: " + src_f.getName());
   return;
   }
   else if(dst_f.exists())
   {
   System.out.println("destination file " + dst_f.getName()
                    + " already exsits");
```

```
   return;
   }

   try
   {
   FileInputStream src = new FileInputStream(src_f);
   FileOutputStream dst = new FileOutputStream(dst_f);
   byte[] buffer = new byte[512];

   while(true) {
     int count = src.read(buffer);
     if(count == -1) break;
     dst.write(buffer,0,count);
   }

   src.close();
   dst.close();
   } catch(IOException e) {
       System.out.println("error during copy: " + e.getMessage());
       if(dst_f.exists())  dst_f.delete();
   }
   }
}
```

8.5 Serialization

When we save object information to a file without *serialization*, we can only save the ASCII version of the data held by the object. We have to reconstruct the data into objects when we read the text file, which means we have to indicate that certain data is associated with certain objects and that certain objects are related. This is time consuming and error prone. Java 1.1 onwards supports the powerful feature of object serialization which allows to serialize objects of any class which implements the `Serializable` interface. Furthermore if our class subclasses other classes, all inherited data fields are serialized automatically. Serialization allows to create persistent objects, that is, objects that can be stored in a file and then reconstituted for later use. For example, if we want to use an object with a program and then use the object again with a later invocation of the same program. Declaring a member `transient` tells Java that this member should not be serialized.

In our example we store string `s1` but not string `s2`.

```
// StringsToStore.java

import java.io.*;

public class StringsToStore implements Serializable
{
   private String s1;
   private transient String s2;

   public StringsToStore(String s1,String s2)
   {
   this.s1 = s1;
   this.s2 = s2;
   }
}
```

The class `StringsToStore` is now used in class `Transient`.

```
// Transient.java

import java.io.*;

public class Transient
{
   public static void main(String[] args)
   {
   try
   {
   FileOutputStream fos = new FileOutputStream("string.ser");
```

```
   ObjectOutputStream oos = new ObjectOutputStream(fos);
   oos.writeObject(new StringsToStore("One","Two"));
   oos.close();
   System.out.println("string.ser saved");
   }
   catch(IOException e)
   {
   System.out.println("could not write file" + e);
   }
   }
}
```

In a more complex example we consider to store the information about a book. We have three files

`Book.java` `WriteBook.java` `ReadBook.java`

The file `Book.java` is given by

```
// Book.java

import java.io.*;

public class Book implements Serializable
{
   public String title;
   public String author;
   public int ISBN;

   public Book(String title,String author,int ISBN)
   {
   this.title = title;
   this.author = author;
   this.ISBN = ISBN;
   }

   public String toString()
   {
   return new String(title + "," + author + "," + ISBN);
   }
}
```

The file `WriteBook.java` is given by

```
// WriteBook.java

import java.io.*;

class WriteBook
{
   public static void main(String[] args)
   {
   Book b = new Book("SymbolicC++","Steeb",1852332603);
   ObjectOutputStream oos = null;
   try
   {
   oos = new ObjectOutputStream(new FileOutputStream("Book.ser"));
   oos.writeObject(b);
   }

   catch(Exception e)
   {
   e.printStackTrace();
   }

   finally
   {
   if(oos != null)
   {
   try { oos.flush(); }
   catch(IOException ioe) { }
   try { oos.close(); }
   catch(IOException ioe) { }
   } // end if
   } // end finally

   } // end main
} // end class
```

The file `ReadBook.java` is given by

```
// ReadBook.java

import java.io.*;

class ReadBook
{
   public static void main(String[] args)
   {
   ObjectInputStream ois = null;
   try
   {
   ois = new ObjectInputStream(new FileInputStream("Book.ser"));
   Object o = ois.readObject();
   System.out.println("Read object " + o);
   }
   catch(Exception e)
   {
   e.printStackTrace();
   }

   finally
   {
   if(ois != null)
   {
   try
   {
   ois.close();
   }
   catch(IOException ioe)
   { }
   } // end if
   } // end finally

   } // end main
}  // end class
```

An `ObjectOutputStream` writes basic data types and graphs of Java objects to an `OutputStream`. The objects can be read (reconstructed) using an

`ObjectInputStream`

Persistent storage of objects can be accomplished by using a file for the stream.

The class `Book` represents data we would like to make persistent. We want to archive it to disk and reload in a later session. We had to declare that the `Book` class implements the

`java.io.Serializable`

interface. The `Serializable` interface does not have any methods. It is simply a signal interface that indicates to the Java virtual machine that we want to use the default serialization mechanism.

We first compile the file `Book.java`

```
javac Book.java
```

to get the file `Book.class`. Then we compile the file `WriteBook.java`

```
javac WriteBook.java
```

to get the file `WriteBook.class`. Then we run this file. `WriteBook` creates an

`ObjectOutputStream`

for the `Book` object and writes it to a `FileOutputStream` named `Book.ser`. This means it formats the object as a stream of bytes and saves it in the file `Book.ser`. Next we compile the file `ReadBook.java`

```
javac ReadBook.java
```

to get the file `ReadBook.class`. Then we run this file

```
java ReadBook
```

This creates an `ObjectInputStream` from the `FileInputStream`, `Book.ser`. In other words, it reads the byte stream from `Book.ser` and reconstitutes the `Book` object from it. `ReadBook` then prints the object. The output is

```
Read object SymbolicC++,Steeb,1852332603
```

The meaning of the modifier `transient` is that it is not part of the persistent state of an object.

The extension to arrays is given in the next programs. We also add as an other attribute to `Book` the price as a floating point number.

```
// Book.java

import java.io.*;

public class Book implements Serializable
{
   public String title;
   public String author;
   public float price;

   public Book(String title,String author,float price)
   {
   this.title = title;
   this.author = author;
   this.price = price;
   }

   public String toString()
   {
   return new String(author + "," + title + "," + price);
   }
}
```

```
// WriteBook.java

import java.io.*;
import Book;

class WriteBook
{
   public static void main(String[] args)
   {
   Book[] b = new Book[2];
   b[0] = new Book("SymbolicC++","Steeb",(float) 123.45);
   b[1] = new Book("Workbook","Willi",(float) 45.60);
   ObjectOutputStream oos = null;
   try
   {
   oos = new ObjectOutputStream(new FileOutputStream("Book.ser"));
   oos.writeObject(b[0]);
   oos.writeObject(b[1]);
   }

   catch(Exception e)
   {
   e.printStackTrace();
   }

   finally
   {
   if(oos != null)
   {
   try { oos.flush(); }
   catch(IOException ioe) { }
   try { oos.close(); }
   catch(IOException ioe) { }
   } // end if
   } // end finally

   } // end main
} // end class
```

```
// ReadBook.java

import java.io.*;

class ReadBook
{
   public static void main(String[] args)
   {
   ObjectInputStream ois = null;
   try
   {
   ois = new ObjectInputStream(new FileInputStream("Book.ser"));
   Object[] o = new Object[2];
   o[0] = ois.readObject();
   o[1] = ois.readObject();
   for(int i=0; i < o.length; i++)
   System.out.println("Read object: " + o[i]);
   }
   catch(Exception e)
   {
   e.printStackTrace();
   }

   finally
   {
   if(ois != null)
   {
   try
   {
   ois.close();
   }
   catch(IOException ioe)
   { }
   } // end if
   } // end finally

   } // end main
}  // end class
```

8.6 GZIP and ZIP Compression

The class `GZIPInputStream` implements a stream filter for reading compressed data in the GZIP format. It extends `InflaterInputStream` and defines two additional variables and a constant. The

```
protected CRC32 crc
```

variable identifies the `CRC32` checksum of the compressed data. The

```
protected boolean eos
```

variable identifies the end of the input strean. The

```
static int GZIP_MAGIC
```

constant identifies the magic number of the GZIP header. Magic numbers are used to uniquely identify files of a given format. The `GZIPInputStream()` constructor allow the InputStream object and the input buffer size to be specified.

The class `GZIPOutputStream` is the output analog of the `GZIPInputStrean`. The class `GZIPOutputStream` implements a stream filter for writing compressed data in the GZIP format. It extends `DeflaterOutputStream` to support `GZIP` compression. It adds the `crc` field variable to calculate a `CRC32` checksum on the compressed data. The `GZIPOutputStream()` constructor allow the OutputStream object and the buffer size to be specified. Thus these classes are for reading/writing data using GZIP compression/decompression scheme.

The following program shows an example. The file to be compressed is passed on the command line. For example after compiling we enter at the command line

```
java GZIP Graph2D1.java
```

The size of the compressed file `myfile.gz` is 390 bytes compared to the size of 733 bytes of the original file `Graph2D1.java`.

```
// GZIP.java

import java.io.*;
import java.util.zip.*;

public class GZIP
{
   public static void main(String[] args)
   {
   try
```

```
   {
   BufferedReader in1 =
      new BufferedReader(
        new FileReader(args[0]));

   BufferedOutputStream out =
      new BufferedOutputStream(
        new GZIPOutputStream(
          new FileOutputStream("myfile.gz")));

   System.out.println("writing file");
   int c;
   while((c = in1.read()) !=  -1)
     out.write(c);
   in1.close();
   out.close();

   System.out.println("reading file");
   BufferedReader in2 =
      new BufferedReader(
        new InputStreamReader(
         new GZIPInputStream(
          new FileInputStream("myfile.gz"))));

   String s;
   while((s = in2.readLine()) != null)
     System.out.println(s);
   }
   catch(Exception e)
   {
   e.printStackTrace();
   }
   }  // end main
}
```

The classes `ZipInputStream` and `ZipOutputStream` are for reading/writing ZIP archive files. The `ZipFile` class is used to read `.zip` files. The `ZipFile()` constructor opens a ZIP file for reading. A `File` object or a `String` object containing a file name may be provided to the `ZipFile()` constructor. The method `entries()` returns an `Enumeration` object containing the `.zip` file entries. The `ZipEntry` class encapsulates a `.zip` entry. It represents a compressed file that is stored within the `.zip` file. The `DEFLATED` and `STORED` constants are used to identify whether a `.zip` entry is compressed or merely stored as uncompressed data within the `.zip` file. The `ZipEntry()` constructor is used to create a named `.zip` entry.

The next two programs show the application of these classes, their methods and data fields.

```
// MyZip.java
//
// use: java MyZip FileName.zip FileName.txt

import java.io.*;
import java.util.*;
import java.util.zip.*;

public class MyZip
{
 public static void main(String[] args)
 {
  try
  {
   ZipOutputStream f =
       new ZipOutputStream(new FileOutputStream(args[0]));

   for(int i=1;i<args.length;i++)
   {
    System.out.println(args[i]);
    try
    {
     byte[] b = new byte[1];
     ZipEntry ze = new ZipEntry(args[i]);
     FileInputStream is = new FileInputStream(args[i]);

     f.putNextEntry(ze);
     while(is.available()>0)
     {
      is.read(b);
      f.write(b,0,1);
    }
```

```
     f.closeEntry();
     is.close();
    }
    catch(Exception e)
    {
     System.out.println("Exception "+e.toString());
    }
   }
   f.close();
  }
  catch(Exception e)
  {
   System.out.println("Exception "+e.toString());
  }
 }
}

// MyUnzip.java
//
// use: java MyUnzip FileName.zip

import java.io.*;
import java.util.*;
import java.util.zip.*;

public class MyUnzip
{
 public static void main(String[] args)
 {
  try
  {
   byte[] b = new byte[1];
   ZipFile f = new ZipFile(args[0]);
   Enumeration entries = f.entries();
   if(args.length==1)
   for(int i=0;i<f.size();i++)
   {
    try
    {
     ZipEntry ze = (ZipEntry)entries.nextElement();
     InputStream is = f.getInputStream(ze);
     FileOutputStream os = new FileOutputStream(ze.getName());

     System.out.println(ze.toString());
     while(is.available()>0)
```

```
     {
      b[0]=(byte)is.read();
      os.write(b,0,1);
     }
     is.close();
     os.close();
    }
    catch(Exception e)
    {
     System.out.println("Exception "+e.toString());
    }

   }
   else
   for(int i=1;i<args.length;i++)
   {
    try
    {
     ZipEntry ze = f.getEntry(args[i]);
     InputStream is = f.getInputStream(ze);
     FileOutputStream os = new FileOutputStream(ze.getName());

     System.out.println(ze.toString());
     while(is.available()>0)
     {
      b[0] = (byte)is.read();
      os.write(b,0,1);
     }
     is.close();
     os.close();
    }
    catch(Exception e)
    {
     System.out.println("Exception "+e.toString());
    }
   }
  }
  catch(Exception e)
  {
   System.out.println("Exception "+e.toString());
  }
 }
}
```

8.7 JPEG Files

JPEG is short for Joint Photographic Experts Group. This is a group of experts nominated by national standards bodies and major companies to work to produce standards for continuous tone image coding.

JPEG codec encodes bitmaps as JPEG files. JPEG is designed for compressing either full-color or gray-scale images of natural, real-world scenes. JPEG handles only still images, but there is a related standard called MPEG for motion pictures. The fundamental advantage of JPEG is that it stores full color information: 24bits/pixel (16 million colors). GIF, the other image format widely used on the net, can only store 8bits/pixel (256 colors or fewer colors).

The basic steps to save an image as a JPEG file are as follows:

1) Create a `BufferdImage` with the same dimensions as our `Component`.

2) Draw the `Component` into the `BufferedImage`.

3) Save the `BufferedImage` into a file using the JPEG package and `FileOutputStream`.

The following program shows an application. It stores the phase portrait of the Ikeda Laser map as a `jpg` file. The data field `TYPE_INT_RGB` represents an image with 8-bit RGB Color Components packed into integer pixels.

```
// JPEG1.java

import com.sun.image.codec.jpeg.*;
import java.awt.*;
import java.awt.geom.Line2D;
import java.awt.image.BufferedImage;
import java.io.FileOutputStream;

public class JPEG1 extends Frame
{
   BufferedImage bi;
   Graphics2D g2;

   public JPEG1()
   {
   bi = new BufferedImage(400,400,BufferedImage.TYPE_INT_RGB);
   g2 = bi.createGraphics();

   double x = 0.5; double y = 0.5; // initial value
```

```
   double x1, y1;
   double c1 = 0.4, c2 = 0.9, c3 = 9.0;
   double rho = 0.85;
   int T = 20000;
   for(int t=0; t < T; t++)
   {
   x1 = x; y1 = y;
   double taut = c1 - c3/(1.0 + x1*x1 + y1*y1);
   x = rho + c2*x1*Math.cos(taut) - y1*Math.sin(taut);
   y = c2*(x1*Math.sin(taut) + y1*Math.cos(taut));
   double m = 90*x + 200;
   double n = 90*y + 200;
   g2.draw(new Line2D.Double(m,n,m,n));
   }

   try
   {
   FileOutputStream jpegOut = new FileOutputStream("output.jpg");
   JPEGImageEncoder jie = JPEGCodec.createJPEGEncoder(jpegOut);
   jie.encode(bi);
   jpegOut.close();
   System.exit(0);
   }
   catch(Exception e) { }
   } // end constructor JPEG1()

   public static void main(String args[])
   {
   JPEG1 jp = new JPEG1();
   }
}
```

8.8 Internationalization

With *Internationalization* our Java program can be adapted to various languages and regions without code changes. The term Internationalization is abbreviated as I18N, since there are 18 letters between the first I and the last N. Support for new languages does not require recompilation or code changes.

A *Locale object* is an identifier for a particular combination of language and region:

```
en US  (English and USA)
en GB  (English and Great Britian)
fr FR  (French and France)
de DE  (German and Germany)
```

In the following two programs we show how Internationalization works. In the first example we have a `JFrame` with two buttons `Yes` and `No`. We want these buttons to display `Ja` and `Nein` for its German version and `Oui` and `No` for its French version. First we write an ASCII file (text file) with the contents

```
yesMessage=Yes
noMessage=No
```

and save this file as `Messages_en_UK.properties`. The we write another ASCII file for the German version

```
yesMessage=Ja
noMessage=Nein
```

and save it as `Messages_de_DE.properties`. Finally we write the French version

```
yesMessage=Oui
noMessage=No
```

and save it as `Messages_fr_FR.properties`. Notice that `yesMessage` and `noMessage` are the same in the English, German and French files, these words are keys. These keys must no change. After compiling the Java file `I18N.java` we run the program with

```
java I18N de DE
```

or

```
java I18N fr FR
```

and so on.

```
// I18N.java

import java.util.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class I18N extends JFrame
{
   String yesCaption;
   String noCaption;

   static String language;
   static String country;

   JButton yesButton, noButton;

   public I18N()    // default constructor
   {
   Locale locale = new Locale(language,country);
   ResourceBundle captions =
   ResourceBundle.getBundle("Messages",locale);

   yesCaption = captions.getString("yesMessage");
   noCaption = captions.getString("noMessage");

   yesButton = new JButton(yesCaption);
   noButton = new JButton(noCaption);

   getContentPane().add(yesButton,BorderLayout.WEST);
   getContentPane().add(noButton,BorderLayout.EAST);
   }  // end default constructor

   static public void main(String[] args)
   {
   if(args.length != 2)
   {  System.out.println("Usage: java I18N Language country");
      System.out.println("For example: java I18N de DE");
   System.exit(0);
   }

   language = new String(args[0]);
   country = new String(args[1]);
```

```
   I18N frame = new I18N();

   frame.addWindowListener(new WindowAdapter()
   {
   public void windowClosing(WindowEvent e)
   { System.exit(0); }
   });

   frame.setBounds(0,0,200,100);
   frame.setVisible(true);
   }  // end main
}
```

Obviously Internationalization can also be applied in text mode. The following example shows this. The file text `MessagesBundle.properties` contains

```
greetings=Hello
inquiry=How are you?
farewell=Goodbye
```

The text file `MessagesBundle_de_DE.properties` contains

```
greetings=Hallo
inquiry=Wie geht es Ihnen?
farewell=Auf Wiedersehen
```

The text file `MessagesBundle_fr_FR.properties` contains

```
greetings=Bonjour
inquiry=Commet allez-vous?
farewell=Au revoir
```

The Java file is given by

```
// I18N1.java

import java.util.*;

public class I18N1
{
   static public void main(String[] args)
   {
   String language;
   String country;
```

```
   if(args.length != 2)
   {
   language = new String("en");
   country = new String("GB");
   }
   else
   {
   language = new String(args[0]);
   country = new String(args[1]);
   }

   Locale currentLocale;
   ResourceBundle messages;

   currentLocale = new Locale(language,country);
   messages = ResourceBundle.getBundle("MessagesBundle",
                                       currentLocale);

   System.out.println(messages.getString("greetings"));
   System.out.println(messages.getString("inquiry"));
   System.out.println(messages.getString("farewell"));
   } // end main
}
```

8.9 Locking Files for Shared Access

Using the `FileLock` class and `FileChannel` class we can use a file lock to restrict access to a file from multiple processes. We have the option of restricting access to an entire file or just a region of it. A file-lock is either shared or exclusive. A file lock object is created each time a lock is acquired on a file via one of the `lock()` or `tryLock()` methods of the `FileChannel` class. A file-lock is initially valid. It remains valid until the lock is released by invoking the `release()` method. The `release()` method is in the `FileLock` class. The following two programs show an example.

We compile the program `LockFile.java` which accesses the file `data.dat` for read and write. Then we run the program. The program gets an exclusive lock on the file `data.dat`, reports when it has the lock, and then waits until we press the Enter key. Before we press the Enter key we start a new process by compiling and running the program `NoOfLines.java`. This program counts the numbers of lines in the file `data.dat`. Since the file `data.dat` is locked it cannot access the file only after we press the Enter key in the first process. Then the lock is released.

```
// LockFile.java

import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class LockFile
{
   public static void main(String[] args) throws IOException
   {
   File f = new File("data.dat");
   RandomAccessFile raf = new RandomAccessFile(f,"rw");
   FileChannel channel = raf.getChannel();
   FileLock lock = channel.lock();
   try
   {
   System.out.println("locked");
   System.out.println("Press ENTER to continue");
   System.in.read(new byte[10]);
   }
   finally
   {
   lock.release();
   }
   }
}
```

```
// NoOfLines.java

import java.io.*;
import java.util.*;

public class NoOfLines
{
   public static void main(String[] args) throws IOException
   {
   FileInputStream fin = new FileInputStream("data.dat");
   BufferedReader in;
   in = new BufferedReader(new InputStreamReader(fin));

   int count = 0;

   while(in.readLine() != null)
   {
   count++;
   }
   System.out.println("count = " + count);

   } // end main
}
```

8.10 Security API, Signature and DSA Algorithm

The Digitial Signature Standard (DSS), created by the NIST, specifies DSA (Digital Signature Algorithm) as the algorithm for digital signatures and SHA-1 (Secure Hash Algorithm) for hashing. DSA is for signatures only and is not an encryption algorithm. The DSA has three components:

1) key generation

2) signature creation

3) signature verification

DSA is a public key algorithm; the secret key operates on the message hash generated by SHA-1; to verify a signature, one recomputes the hash of the message, uses the public key to decrypt the signature and then compare the results. The key size is variable from 512 to 1024 bits which is adequate for current computing capabilities as long as we use more than 768 bits. Signature creation is roughly the same speed as with RSA, but is 10 to 40 times as slow for verification. However, these numbers depend partially on the assumptions made by the benchmarker. Since verification is more frequently done than creation, this is an issue worth noting.

The only known cracks (forgery) are easily circumvented by avoiding the particular moduli (prime factor of $p-1$ where p is the public key) that lead to weak signatures. DSS is less susceptible to attacks than RSA; the difference is that RSA depends on a secret prime while DSA depends on a public prime. The verifier can check that the prime number is not a fake chosen to allow forgery. It is possible to implement the DSA algorithm such that a "subliminal channel" is created that can expose key data and lead to forgable signatures so one is warned not to used unexamined code.

Java provides all the tools of the DSA. The

```
KeyPairGenerator
```

is used to generate pairs of public and private keys. Key pairs generators are constructed using the `getInstance()` factory method. The `Signature` class is used to provide applications the functionality of a digital signature algorithm. Digital signature are used for the authorization and integrity assurance of digital data. The method

```
byte[] sign()
```

returns the signature bytes of all the data updated. The method

```
void initSign(PrivateKey)
```

initializes this object for signing. The interface `PrivateKey` merely serves to group (and provide type safety for) all private key interfaces. The method

```
boolean verify(byte[])
```

verfies the passed in signature.

For the following example assume that we have a text file `data.dat` containing the line

```
amount: 2000
```

If we run the program with

```
java MySignature data.dat
```

then the program stops and waits for input from the keyboard. If we change at this stage the file `data.dat`, for example to

```
amount: 3000
```

then after entering an arbitrary string we obtain the output

```
signature verifies: false
```

Obviously if we do not change the file the output will be

```
signature verifies: true
```

The program is

```
// MySignature.java

import java.io.*;
import java.security.*;

class MySignature
{
   public static void main(String[] args)
   {
   try {
     // generate a key pair
     KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA");
     keyGen.initialize(1024,new SecureRandom());
     KeyPair pair = keyGen.generateKeyPair();

     // create a Signature object
     // to use for signing and verify
     Signature dsa = Signature.getInstance("SHA/DSA");

     // initialize the Signature object for signing
     PrivateKey priv = pair.getPrivate();
     dsa.initSign(priv);

     // Update and sign the data
     FileInputStream fis = new FileInputStream(args[0]);
     byte b;
     while(fis.available() != 0) {
            b = (byte) fis.read();
            dsa.update(b);
     } // end while
     fis.close();

     // sign it
     byte[] sig = dsa.sign();

     // to stop the system we read from
     // the keyboard an arbitrary string
     System.out.print("enter string: ");
     BufferedReader kbd =
        new BufferedReader(new InputStreamReader(System.in));
     String str = kbd.readLine();

     // verify the signature
```

```
      PublicKey pub = pair.getPublic();
      dsa.initVerify(pub);

      // update and verify the data
      fis = new FileInputStream(args[0]);

      while(fis.available() != 0) {
        b = (byte) fis.read();
        dsa.update(b);
      } // end while

      fis.close();
      boolean verifies = dsa.verify(sig);
      System.out.println("signature verifies: " + verifies);

      } // end try block
      catch(Exception e) {
        System.err.println("Caught exceptio " + e.toString());
      }

   } // end main
}
```

Chapter 9

Threads

9.1 Introduction

Multitasking and *multithreading* are two types of concurrencies. Multitasking refers to the ability of executing more than one program at the time. It is usually supported on the operating system level. Multithreading, on the other hand, refers to a single pogram which has more than one execution thread running concurrently. Each of these independent subtasks is called a *thread*. An execution thread refers to the execution of a single sequence of instructions. In Java support for multithreaded programmming is built into the language.

Threads are sequences of code that can be executed independently alongside one another. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently. Thus a thread is an independent sequential flow of control within a process. Threads run with programs. A single application or applet can have many threads doing different things independently. The `Thread` class is defined in `java.lang` as a subclass of the `Object` class. To use Threads in a program, one needs to define a local subclass of the `Thread` class and therein override its `void run()` method. We put the code that we want the threads of that subclass to execute in that `void run()` method.

The `public abstract interface Runnable` should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method of no arguments called `run()`.

There are two ways in which to initiate a new thread of execution. The first is to create a subclass of the `Thread` class and redefine the `run()` method to perform the set of actions that the thread is intended to do. The second one is to define a new class that implements the `Runnable` interface. It must define the `run()` method and can be started by passing an instance of the class to a new `Thread` instance. In both cases we define a class which specifies the operations or actions that the new thread (of execution) performs.

A thread can be in any one of four states:

1) **New**: the thread object has been created but it has not been started yet so it cannot run.

2) **Runnable**: this means that a thread can be run when the time-slicing mechanism has CPU cycles available for the thread. Thus, the thread might or might not be running, but there is nothing to prevent it from being run if the scheduler can arrange it. It is not dead or blocked.

3) **Dead**: the normal way for a thread to die is by returning from its `run()` method.

4) **Blocked**: the thread could be run but there is something that prevents it. While a thread is in the blocked state the scheduler will simply skip over it and not give any CPU time. Until a thread re-enters the runnable state it will not perform any operations. A thread can be blocked for five reasons. First we have put the thread to sleep by calling the method `sleep(milliseconds)`, in which case it will not be run for the specified time. Second we have suspended the execution of the thread by calling the method `suspend()`. It will not become runnable again until the thread gets the `resume()` message. Third we have suspended the execution of the thread with the method `wait()`. The method

```
public final void wait() throws InteruptedException
```

waits to be notified by another thread of a change in this object. It will not become runnable again until the thread gets the `notify()` or `notifyAll()` message. The method

```
public final native void notify()
```

wakes up a single thread that is waiting on this object's monitor. A thread waits on an object's monitor by calling one of the wait methods. The method

```
public final native void notifyAll()
```

wakes up all threads that are waiting on this object's monitor. A thread waits on an object's monitor by calling one of the wait methods. Fourth the thread is waiting for some Input/Output to complete. Finally the thread is trying to call a `synchronized` method on another object and that object's lock is not available. The keyword `synchronized` indicates that while a method is accessing an object, other `synchronized` methods are blocked from accessing that object.

9.2 Thread Class

The `Thread` class provides all the facilities required to create a class which can execute within its own lightweight process (within the JVM). Next we give the methods that can change the state of a thread. The

```
void start()
```

method in class `Thread` causes this thread to begin execution. The Java Virtual Machine calls the `run()` method of this thread. The `Runnable` interface has a single method called `run()`. The class which implements the `Runnable` interface must therefore supply its own `run()` method. Starting a thread causes the `run()` method to be executed. This method is executed for a brief time and then another thread of the application is executed. This thread runs briefly and is then suspended so another thread can run and so on. If this thread was constructed using a separate `Runnable` object, then that `Runnable` object's `run()` method is called; otherwise, this method does nothing and returns. The method

```
static void sleep(long millis)
```

in the `Thread` class causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds. The method

```
void yield()
```

causes the currentlty executing thread object to temporarily pause and allow other threads to execute. The method

```
void destroy()
```

destroys this thread without any cleanup.

There are a number of methods in the class `Thread` that provide information about the status of the process. The method

```
boolean isAlive()
```

tests to see if the thread has started execution. On the other hand the method

```
boolean isInterrupted()
```

tests if this thread has been interrupted.

Every thread has a unique name. If we do not provide one, then a name is automatically generated by the system. There are two methods that access and set a thread's name. The method

```
String getName() // in Thread class
```

returns this thread's name. The method

```
void setName(String name)
```

changes the name of this thread to be equal to the argument `name`.

The field

```
static int MAX_PRIORITY
```

is an integer which defines the maximum priority of a thread. The field

```
static int MIN_PRIORITY
```

is an integer which defines the minimum priority of a thread.

The methods

```
void resume(),   void stop(),   void suspend()
```

are deprecated. Instead of using the `stop()` method, it is recommended that threads monitor their execution and stop by returning from their `run()` method.

In many cases we call the method

```
void repaint()
```

inside the `run()` method. The method `repaint()` is in class `Component`. It repaints this component, i.e. it calls the method `paint()`. The method `paint()` is in class `Component`. It paints this component. The method

```
void repaint(long tm)
```

repaints the component. This will result in a call to `update()` within `tm` milliseconds, i.e. `tm` is the maximum time in milliseconds before update.

9.3 Examples

In our first two examples we implement two balls one moving in x direction the other in y direction. In the first program `for` loops are used to pass the time. In the second program we use the `Calendar` class.

```
// MoveBalls1.java

import java.awt.Graphics;
import java.awt.*;
import java.applet.Applet;

public class MoveBalls1 extends Applet implements Runnable
{
   int x = 0, y = 0;

   public void init()
   {
   Thread thread = new Thread(this);
   thread.start();
   }

   public void run()
   {
   int i, j, k;
   for(j=0; j < 200; j++)
   {
   x++; y++;
   repaint(50);
   for(i=0; i < 1000; i++) { for(k=0; k < 1000; k++) {   } }
   }
   for(j=0; j < 200; j++)
   {
   x--; y--;
   repaint(50);
   for(i=0; i < 1000; i++) { for(k=0; k < 1000; k++)  {   } }
   }
   } // end run()

   public void paint(Graphics g)
   {
   g.setColor(Color.blue);   g.drawOval(x,100,20,20);
   g.setColor(Color.red);    g.drawOval(100,y,20,20);
   }  // end paint(Graphics g)
}  // end class MoveBalls1
```

```
// MoveBalls2.java

import java.awt.Graphics;
import java.awt.*;
import java.applet.Applet;
import java.util.Calendar;

public class MoveBalls2 extends Applet implements Runnable
{
   int d = 0;

   public void init()
   {
   Thread thread = null;
   if(thread == null)
   {
   thread = new Thread(this);
   thread.start();
   }
   }

   public void run()
   {
   for(;;)
   {
   Calendar now = Calendar.getInstance();
   d = now.get(Calendar.MILLISECOND);
   repaint(5);
   } // end for loop
   } // end run

   public void paint(Graphics g)
   {
   d = d + 5;
   g.setColor(Color.red);   g.fillOval(d,25,50,50);
   g.setColor(Color.blue);  g.fillOval(25,d,50,50);
   }  // end paint(Graphics g)
}  // end class MoveBalls2
```

In our third example we define a new class called `Counter` that implements the `Runnable` interface. It defines a `run()` method and will be started by passing an instance of the class to a new `Thread` instance.

```
// Counter.java

import java.awt.*;
import java.applet.*;

public class Counter extends Applet implements Runnable
{
   Thread timer = null;
   int count = 0;

   public void start()
   {
     if(timer == null)
     {
     timer = new Thread(this);
     timer.start();
     }
   }

   public void run()
   {
   while(timer != null)
   {
     try  { Thread.sleep(1000); }
     catch(Exception e) { }
     count++;
     repaint();
     }
   }

   public void paint(Graphics g)
   {
   g.setFont(new Font("SanSerif",Font.BOLD,24));
   g.drawString("Count = " + count,10,20);
   }
}
```

The HTML file is given by

```
<HTML>
<COMMENT> Counter.html </COMMENT>

<APPLET CODE="Counter.class" width=275 height=135>
</APPLET>
</HTML>
```

In our fourth example we create a subclass `TestThread` of class `Thread` and redefine the `run()` method to perform the set of actions that the thread is intended to do. The names of the threads are 0, 1, 2, 3.

```
// MyThread.java

class TestThread extends Thread
{
   int sleepTime;

   public TestThread(String s)
   {
   super(s);
   sleepTime = (int) (500*Math.random());
   System.out.println("Name: " + getName() + "\t Sleep: " + sleepTime);
   }  // end constructor TestThread

   public void run()
   {
   try {  sleep(sleepTime); }
   catch(Exception e) { }
   System.out.println("Thread " + getName());
   }  // end run
}  // end class TestThread

public class MyThread
{
   public static void main(String[] args)
   {
   TestThread thread0, thread1, thread2, thread3;
   thread0 = new TestThread("0"); thread1 = new TestThread("1");
   thread2 = new TestThread("2"); thread3 = new TestThread("3");
   thread0.start();
   thread1.start();
   thread2.start();
   thread3.start();

   }  // end main
}     // end class MyThread
```

A typical output is

```
first run:
Name: 0  Sleep: 223
Name: 1  Sleep:  55
Name: 2  Sleep: 401
Name: 3  Sleep: 482
Thread 0
Thread 2
Thread 3
Thread 1

second run
Name: 0  Sleep:  36
Name: 1  Sleep: 145
Name: 2  Sleep: 345
Name: 3  Sleep: 290
Thread 0
Thread 3
Thread 1
Thread 2
```

9.4 Priorities

The *priority* of a thread tells the scheduler how important this thread is. If there are a number of threads blocked and waiting to be run, the scheduler will run the one with the highest priority first. However, this does not mean that threads with lower priority do not get run. This means we cannot deadlocked because of priorities. Lower priority threads just tend to run less often. In Java we can read the priority of a thread with the method `getPriority()` and change it with the method `setPriority()`. The following program shows an application of these methods.

```
// Bounce.java

import java.awt.*;
import java.awt.event.*;

public class Bounce extends Frame
                    implements WindowListener, ActionListener
{
   Canvas canvas;

   public Bounce()
   {
   setTitle("BouncingBalls");
   addWindowListener(this);

   canvas = new Canvas();
   add("Center",canvas);

   Panel panel = new Panel();
   add("North",panel);

   Button b = new Button("Normal Ball");
   b.addActionListener(this);
   panel.add(b);

   b = new Button("High Priority Ball");
   b.addActionListener(this);
   panel.add(b);

   b = new Button("Close");
   b.addActionListener(this);
   panel.add(b);
   }  // end constructor Bounce()
```

```
   public void windowClosing(WindowEvent e) { System.exit(0); }
   public void windowClosed(WindowEvent e) { }
   public void windowOpened(WindowEvent e) { }
   public void windowDeiconified(WindowEvent e) { }
   public void windowIconified(WindowEvent e) { }
   public void windowActivated(WindowEvent e) { }
   public void windowDeactivated(WindowEvent e) { }

   public void actionPerformed(ActionEvent action)
   {
   if(action.getActionCommand() == "Normal Ball")
   {
   Ball ball = new Ball(canvas,Color.blue);
   ball.setPriority(Thread.NORM_PRIORITY);
   ball.start();
   }
   else if(action.getActionCommand() == "High Priority Ball")
   {
   Ball ball = new Ball(canvas,Color.red);
   ball.setPriority(Thread.NORM_PRIORITY+1);
   ball.start();
   }
   else if(action.getActionCommand() == "Close")
   System.exit(0);
   }

   public static void main(String[] args)
   {
   Frame frame = new Bounce();
   frame.setSize(400,300);
   frame.setVisible(true);
   }

}  // end class Bounce

class Ball extends Thread
{
   Canvas box;
   private static final int diameter = 10;
   private int x = 0;
   private int y = 0;
   private int dx = 2;
   private int dy = 2;
   Color color;
```

```
   Ball(Canvas canvas,Color col)
   {
   box = canvas;
   color = col;
   }

   public void draw()
   {
   Graphics g = box.getGraphics();
   g.setColor(color);
   g.fillOval(x,y,diameter,diameter);
   g.dispose();
   } // end draw

   public void move()
   {
   Graphics g = box.getGraphics();
   g.setColor(color);
   g.setXORMode(box.getBackground());
   g.fillOval(x,y,diameter,diameter);
   x += dx;
   y += dy;
   Dimension d = box.getSize();
   if(x < 0) { x = 0; dx = -dx; }
   if(x+diameter >= d.width) { x = d.width-diameter; dx = -dx; }
   if(y < 0) { y = 0; dy = -dy; }
   if(y+diameter >= d.height) { y = d.height - diameter; dy = -dy; }
   g.fillOval(x,y,diameter,diameter);
   g.dispose();
   } // end move

   public void run()
   {
   draw();
   for(;;)
   {
   move();
   try { sleep(10); }
   catch(InterruptedException e) { }
   } // end for loop
   } // end run

}  // end class Ball
```

9.5 Synchronization and Locks

Threads behave fairly independently with the Java virtual machine switching between the threads. This can cause problems since it is not known when a thread will be paused to allow another thread to run. Java has built-in support to prevent collisions over one kind of resource: the memory in an object. We typically make the data elements of a class `private` and access that memory only through methods, we can prevent collisions by making a particular method `synchronized`. Only one thread at the time can call a `synchronized` method for a particular object. Thus to prevent one thread from interfering with the other the `synchronized` modifier can be used when defining a method.

Any method can be preceded by the word `synchronized`. The rule is: no two threads may be executing `synchronized` methods of the same object at the same time. The Java system enforces this rule by associating a monitor lock with each object. When a thread calls a `synchronized` method of an object, it tries to grab the object's monitor lock. If another thread is holding the lock, it waits until that thread releases it. A thread releases the monitor lock when it leaves the `synchronized` method. Of one `synchronized` method of a call contains a call to another, a thread may have the same lock multiple times. Java keeps track of that correctly.

Thus there is a lock with every object. The `synchronized` statement computes a reference to an object. It then attempts to perform a lock operation on that object and does not proceed further until the lock operation has successfully completed. A lock operation may be delayed because the rules about locks can prevent the main memory from participating until some other thread is ready to perform one or more unlock operations. After the lock operation has been performed, the body of the `synchronized` statement is executed. Normally, a compiler ensures that the lock operation implemented by a monitorenter instruction executed prior to the execution of the body of the `synchronized` statement is matched by an unlock operation implemented by a monitorexit instruction whenever the `synchronized` statement completes, whether completion is normal or abrupt. The Java Virtual Machine provides separate monitorenter and monitorexit instructions that implements the lock and unlock operations.

A `synchronized` method automatically performs a lock operation when it is invoked. Its body is not executed until the lock operation has successfully completed. If the method is an instance method, it locks the lock associated with the instance for which it was invoked. This means, the object that will be known as `this` during execution of the method's body. If the method is `static`, it locks the lock associated with the `Class` object that represents the class in which the method is defined. If execution of the method's body is ever completed, either normally or abruptly, an unlock operation is automatically performed on that same lock.

In the following program we start two threads `f1` and `f2`. The method `display()` is synchronized. The output the program `Synchronized.java` is

```
100 101 102
100 101 102
100 101 102
100 101 102
100 101 102
100 101 102
```

If we change the line

```
synchronized void display()
```

in the program to

```
void display()
```

the output is

```
100 100 101 101 102 102
100 100 101 101 102 102
100 100 101 101 102 102
```

```
// Synchronized.java

public class Synchronized
{
    Count f1, f2;

    Synchronized()
    {
    f1 = new Count(this);
    f2 = new Count(this);
    f1.start();
    f2.start();
    } // end constructor Synchronized

    synchronized void display()
    {
    System.out.print("100 ");
    System.out.print("101 ");
    System.out.print("102 ");
    System.out.println();
    }

    public static void main(String[] args)
    {
    new Synchronized();
    }
}  // end class Synchronized

class Count extends Thread
{
   Synchronized current;

   Count(Synchronized thread)
   {
   current = thread;
   }

   public void run()
   {
   int i;
   for(i=0; i < 3; i++)
   current.display();
   }
}
```

In the followings programs `BankAccount.java` and `Transfer.java` we consider the problem of transferring money between bank accounts. It is left as an exercise to the reader to investigate whether or not the method `getBalance()` should be synchronized or not in the following two programs.

```
// BankAccount.java

public class BankAccount
{
   private double balance;

   public BankAccount(double balance)
   {
   this.balance = balance;
   }

   public double getBalance()
   {
   return balance;
   }

   public synchronized void deposit(double amount)
   {
   balance += amount;
   }

   public synchronized void withdraw(double amount)
   throws RuntimeException
   {
   if(amount > balance)
   {
   throw new RuntimeException("Overdraft");
   }
   balance -= amount;
   }

   public synchronized void transfer(double amount,BankAccount destination)
   {
   this.withdraw(amount);
   Thread.yield();
   destination.deposit(amount);
   }
}  // end class BankAccount
```

```
// Transfer.java

import BankAccount;

public class Transfer implements Runnable
{
   public static void main(String[] args)
   {
   Transfer x =
   new Transfer(new BankAccount(100.0),new BankAccount(100.0),50.0);
   Thread t = new Thread(x);

   t.start();
   Thread.yield(); // the thread on which yield() is invoked would
                   // move from running state to ready state

   System.out.println("Account A has Dollar: " + x.A.getBalance());
   System.out.println("Account B has Dollar: " + x.B.getBalance());
   }

   public BankAccount A, B;
   public double amount;

   public Transfer(BankAccount A,BankAccount B,double amount)
   {
   this.A = A;
   this.B = B;
   this.amount = amount;
   }

   public void run()
   {
   System.out.println("Before transfer A has Dollar: "
                      + A.getBalance());
   System.out.println("Before transfer B has Dollar: "
                      + B.getBalance());
   A.transfer(amount,B);
   System.out.println("After transfer A has Dollar: "
                      + A.getBalance());
   System.out.println("After transfer B has Dollar: "
                      + B.getBalance());
   }
}
```

9.6 Producer Consumer Problem

The producer consumer problem is a classic *synchronization problem*. The producer and consumer processes share a common buffer. The producer executes a loop in which it puts new items into the buffer and the consumer executes a loop in which it removes items from the buffer. The following important conditions have to be satisfied by the producer and consumer. At most one process (producer or consumer) may be accessing the shared buffer at any time. This condition is called mutual exclusion. When the buffer is full, the producer should be put to sleep. It should only wake up when an empty slot becomes available. This is called synchronization. When the buffer is empty, the consumer should be put to sleep. It should only wake up when at least one slot becomes full. This is also called synchronization.

The following four programs

```
Producer.java
Consumer.java
Buffer.java
Main.java
```

give a solution to this problem. The methods `wait()` and `notify()` are in the class `Object`. Class `Object` is the root of the class hierarchy. Every class has `Object` as a superclass. All objects, including arrays, implement the methods of this class. The method `wait()` causes the current thread to wait until another thread invokes the `notify()` method or the `notifyAll()` method for this object. The method `notify()` wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakend. The choice is arbitrary and occurs at the discretion of the implementation. A thread waits on an object's monitor by calling one of the `wait` methods.

```
// Producer.java

public class Producer extends Thread
{
   private Buffer buffer;

   public Producer(Buffer b)
   {
   buffer = b;
   }

   public void run()
   {
   for(int i=0; i < 10; i++)
   {
```

```
   buffer.put(i);
   System.out.println("Producer put: " + i);

   try
   {
   sleep((int) (Math.random()*100));
   }
   catch(InterruptedException e) { }
   }
   }
}


// Consumer.java

public class Consumer extends Thread
{
   private Buffer buffer;

   public Consumer(Buffer b)
   {
   buffer = b;
   }

   public void run()
   {
   int value = 0;
   for(int i=0; i < 10; i++)
   {
   value = buffer.get();
   System.out.println("Consumer got: " + value);
   }
   }
}


// Buffer.java

public class Buffer
{
   private int contents;
   private boolean available = false;

   public synchronized int get()
   {
```

```
   while(available == false)
   {
   try
   {
   wait();
   }
   catch(InterruptedException e) {  }
   }
   available = false;
   notify();
   return contents;
   }

   public synchronized void put(int value)
   {
   while(available == true)
   {
   try
   {
   wait();
   }
   catch(InterruptedException e) {  }
   }
   contents = value;
   available = true;
   notify();
   }
}


// Main.java

public class Main
{
   public static void main(String[] args)
   {
   Buffer b = new Buffer();
   Producer p = new Producer(b);
   Consumer c = new Consumer(b);
   p.start();
   c.start();
   }
}
```

9.7 Deadlock

Deadlock is a mutual starvation between at least two threads that cannot satisfy each other. We cause deadlock between two threads that are both trying to acquire locks for the same two resources. To avoid this sort of deadlock when locking multiple resources, all threads should always acquire their locks in the same order. There are two resource objects (strings) we will try to get locks for. The first thread tries to lock resource1 then resource2. The second thread tries to lock resource2 and then resource1.

```
// MyDeadLock.java

public class MyDeadLock
{
  public static void main(String[] args)
  {
    final String resource1 = "resource1";
    final String resource2 = "resource2";

    // first thread
    Thread t1 = new Thread() {
      public void run() {
        // Lock resource 1
        synchronized(resource1) {
          System.out.println("Thread 1: locked resource 1");
          // Pause for a bit something.
          // We give the other thread a chance to run.
          // Threads and deadlock are asynchronous things,
          // but we are trying to force deadlock to happen here.
          try {
            Thread.sleep(50);
          } catch(InterruptedException e) { }

          // Now wait till we can get a lock on resource 2
          synchronized(resource2){
            System.out.println("Thread 1: locked resource 2");
          }
        }
      }
    };

    // second thread
    Thread t2 = new Thread() {
      public void run() {
        // This thread locks resource 2 right away
```

```
        synchronized(resource2) {
          System.out.println("Thread 2: locked resource 2");
          // Then it pauses, for the same reason as the first
          // thread does
          try {
          Thread.sleep(50);
      } catch(InterruptedException e) {}

          // Then it tries to lock resource1.
          // However Thread 1 locked resource1, and
          // will not release it till it gets a lock on resource2.
          // This thread holds the lock on resource2, and will not
          // release it till it gets resource1.
          // Thus neither thread can run, and the program freezes up.
          synchronized(resource1) {
            System.out.println("Thread 2: locked resource 1");
          }
        }
      }
    };

    // Start the two threads.
    // Deadlock can occur, and the program will never exit.
    t1.start();
    t2.start();
  }
}
```

Running the program we typically get

```
Thread 1: locked resource 1
Thread 2: locked resource 2
```

and then the program freezes up. Using the debugger `jdb` and `run` we get

```
Thread 2: locked resource 2
Thread 2: locked resource 1
Thread 1: locked resource 1
Thread 1: locked resource 2
```

Chapter 10

Animation

10.1 Introduction

Animation involves changing a picture over and over to simulate movement of some sort. There are several different types of animation we can perform in a Java applet. We can display a sequence of images, or we can display a single image while changing a portion of it. We can change an image by running it through a filter of by changing the colours in the image. We can also perform animation with the basic graphics classes.

Animation is a powerful technique for applets. Having a portion of our display that moves can make our application or our Web page much livelier.

Animation is frequently used in video games. In facet, it is probably one of the most common factors in computer games. Modern adventure games often use sequences of real-life images to give the game a modern feel. Arcade games frequently employ graphical animation, although some have begun to integrate images into the games as well.

Beyond gaming, animation is an excellent tool for computer-assisted learning. Rather than describing a technique with plain old text, we can demonstrate it through animation. This type of animation is often done with images but not necessarily. We may be demonstrating a technique that requires a graphical representation that is computed while the program is running.

To perform animation, we have to create a thread that repeatedly changes the animation sequence and repaints the picture. Because the different animation techniques all use this same method, we can use the same mechanism to trigger the next animation sequence.

Like Web browsers, Java is designed to handle graphics files in two formats - GIF (including animated GIFs) with extension `.gif` and JPEG (with either a `.jpeg` or a `.jpg` extension).

```
jpg-file: JPEG bitmaps
gif-file: GIF CompucServe bitmaps
```

JPEG stands for Joint Photographic Experts Group. Java cannot cope with the standard Windows formats BMP and PCX. If we want to create our own images, we need a suitable graphics application, such as Paint ShopPro or CorelDraw. CorelDraw allows then to convert the `.cdr` file into a `.gif` or `.jpg` file. We can use this for drawing new images, or for converting pictures created in Paint ShopPro or CorelDraw or almost anywhere else - into the GIF and JPEG formats.

GIF files are significantly smaller than JPEG files, and therefore faster to load - an important consideration, especially where files have to be transferred over the World Wide Web. The main reason why JPEG files are larger is that they have palettes of

$$256 \cdot 256 \cdot 256 = 16777216$$

colours as opposed to the 256-colour palette of GIFs. We use JPEGs only where we need to show subtle variations of colour over a huge palette - and if the file is going up onto the Web. Many people will be viewing it on 256, or even 16-colour screens.

An animated GIF is a set of images, which are displayed in sequence, with a defined delay between each. They are used in exactly the same way as still images - the animation is entirely built into the image file, so all Java has to do is display it. The files tend to be large, and that the animation speed is defined within the GIF. There are plenty of ready-made animated GIFs freely available on the Web.

JDK 1.1 provided the capability for applets to play audio files that were in the SunAudio (`.au`) format. JDK 1.2 provides a new sound engine that allows audio files to be played by both applets and applications. The sound engine also provides support for the Musical Instrument Digital Interface (MIDI), the Microsoft Windows audio format (WAVE), the Rich Music Format (RMF), and the Audio Interchange File Format (AIFF).

10.2 Image Class and Animation

Images are stored in `Image` objects, with the files loaded into them by the `getImage()` method. The abstract class

```
public abstract class Image extends Objects
```

is the superclass of all classes that represent graphical images. The image must be obtained in a platform-specific manner. The method

```
Image getImage(URL url,String name)
```

in class `Applet` returns an `Image` object that can be painted on the screen. Thus method `getImage()` takes as its parameter a `URL` - Uniform Resource Locator - the standard means of identifying a file anywhere on the Internet or in a computer. The method

```
URL getCodeBase()
```

in class `Applet` gets the base `URL`. This is the `URL` of the applet itself. The easiest way to handle the `URL` is with the method `getCodeBase()`. This works out the `URL` of the directory in which the program is stored. If the image file is in the same directory, then we would simply follow it up with the filename. If the file is in a sub-directory, we also need to add its name. The method `getCodeBase()` will give the right `URL` when the files are on our system and when they are uploaded to our Web space on our access provider's server. The method `getDocumentBase()` returns the `URL` of the Web page in which the applet is embedded.

If the image file is stored in a completely different place from the code, then we have to define the `URL` using the `java.net.URL` class. This throws an exception that we must `catch`. For example

```
import jave.net.*;

try
{
pic = getImage(new URL("http://zeus.rau.ac.za/diploma/globe.gif"));
}
catch(MalformedURLException e)
{  }
...
```

The address is written in full, then converted to a `URL` object. The `catch` block might get an alternative image from a more secure source.

```
// Animate.java
//
// The files
// figure1.jpg figure2.jpg figure3.jpg figure4.jpg
// are in the same directory as the file
// Animate.java

import java.awt.*;
import java.applet.*;

public class Animate extends Applet implements Runnable
{
   int i;
   Thread animator = null;
   Image[] pic = new Image[4];
   int current = 0;

   public void init()
   {
   pic[0] = getImage(getCodeBase(),"figure1.jpg");
   pic[1] = getImage(getCodeBase(),"figure2.jpg");
   pic[2] = getImage(getCodeBase(),"figure3.jpg");
   pic[3] = getImage(getCodeBase(),"figure4.jpg");
   }

   public void start()
   {
   if(animator == null)
   {
   animator = new Thread(this);
   animator.start();
   }
   }

   public void run()
   {

   while(animator != null)
   {
   i += 10;
   if(i > 400) { i = 0; }
   current++;
   if(current > 3) current = 0;
   repaint();  // repaint() calls paint()
```

```
   try
   {
   Thread.sleep(100);
   } // end try

   catch(InterruptedException e) { }
   } // end while

   } // end run()

   public void paint(Graphics g)
   {
   g.drawImage(pic[current],i,100,this);
   }

}
```

The html-file is `Animate.html` is

```
<HTML>
<APPLET CODE="Animate.class" width=275 height=135>
</APPLET>
</HTML>
```

We recall that the method

```
abstract boolean drawImage(Image img,int x,int y,ImageObserver observer)
```

draws as much of the specified image as is currently available. The `ImageObserver` is an asynchronous update interface for receiving notification about `Image` information as the `Image` is constructed.

It is left as an exercise to rewrite the following code using jpg-files to simulate a flashing traffic light.

```
// Traffic.java

import java.awt.*;
import java.applet.*;
import java.awt.Graphics;

public class Traffic extends Applet implements Runnable
{
    Thread animator = null;
    int t = 0;
    Color[] col = new Color[3];

    public void init()
    {
    col[0] = Color.red; col[1] = Color.yellow; col[2] = Color.green;
    } // end init

    public void start()
    {
    if(animator == null)
    {  animator = new Thread(this);  animator.start(); }
    } // end start

    public void run()
    {
    while(animator != null)
    {
    try { Thread.sleep(2000); }
    catch(Exception e) { }
    t++;
    if(t > 2) t = 0;
    repaint();
    }
    } // end run

    public void paint(Graphics g)
    {
    g.setColor(col[t]);
    g.drawOval(20,20,100,100);
    g.fillOval(20,20,100,100);
    } // end paint
}
```

In the following applet we have a background picture (file `Image0.jpg`) and a moving picture (file `Actor.jpg`).

```
// FigMotion.java

import java.awt.*;
import java.applet.*;

public class FigMotion extends Applet implements Runnable
{
    Thread runner;
    Image Buffer;
    Graphics gBuffer;
    Image background, figure2;
    int x;

    public void init()
    {
    Buffer=createImage(getSize().width,getSize().height);
    gBuffer=Buffer.getGraphics();
    background = getImage(getCodeBase(),"Image0.jpg");
    figure2 = getImage(getCodeBase(),"Actor.jpg");
    }

    public void start()
    {
    if(runner == null) { runner = new Thread(this); runner.start(); }
    }

    public void run()
    {
        while(true) {
          try { runner.sleep(15); }
          catch(Exception e) { }
          gBuffer.drawImage(background,0,0,this);
          x++;
          if(x > getSize().width) x=-125;
          gBuffer.drawImage(figure2,x,150,this);
          repaint();
        }
    }

    public void update(Graphics g)  { paint(g); }
    public void paint(Graphics g)   { g.drawImage(Buffer,0,0,this); }
}
```

Another application shows the following program, where a `jpg` picture is rotated and scaled.

```
// Animation.java

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import java.awt.geom.*;
import java.awt.image.*;
import java.util.*;
import java.net.URL;

public class Animation extends JApplet
{
  AnimationCanvas canvas;
  JButton start, stop;
  URL codeBase;

  public void init()
  {
  Container container = getContentPane();
  codeBase = getCodeBase();
  canvas = new AnimationCanvas(codeBase);
  container.add(canvas);

  start = new JButton("Start");
  start.addActionListener(new ButtonListener());
  stop = new JButton("Stop");
  stop.addActionListener(new ButtonListener());

  JPanel panel = new JPanel();
  panel.add(start);
  panel.add(stop);
  container.add(BorderLayout.SOUTH,panel);
  }

  class ButtonListener implements ActionListener
  {
  public void actionPerformed(ActionEvent e)
  {
  JButton temp = (JButton) e.getSource();
  if(temp.equals(start)) { canvas.start(); }
    else if(temp.equals(stop)) { canvas.stop(); }
  }
```

```
  }
}

class AnimationCanvas extends JPanel implements Runnable
{
  Image image;
  Thread thread;
  BufferedImage bi;
  double x, y, xi, yi;
  double scale;
  int rotate;
  int UP = 0, DOWN = 1;
  int scaleDirection;
  URL eiffelURL;

  AnimationCanvas(URL codeBase)
  {
  setBackground(Color.green);

  try {
    eiffelURL = new URL(codeBase,"/home/java/Eiffel_Tower.JPG");
    }
    catch(java.net.MalformedURLException e)
    {
    System.out.println("bad URL");
    }

    image = new ImageIcon(eiffelURL).getImage();
    MediaTracker mt = new MediaTracker(this);
    mt.addImage(image,1);
    try { mt.waitForAll(); }
    catch(Exception e)
     { System.out.println("Exception while loading image"); }

    if(image.getWidth(this)==-1)
     { System.out.println("Make sure the image exists"
        + "(Eiffel_TOWER.JPG) file in the same directory");
       System.exit(0);
     }

     rotate = (int) (Math.random()*360);
     scale = Math.random()*0.8;
     scaleDirection = DOWN;
     xi = 40.0; yi = 40.0;
     }
```

```
public void steps(int w,int h)
{
x += xi; y += yi;
if(x > w) { x = w-1; xi = Math.random()*(-w/32.0); }
if(x < 0) { x = 2.0; xi = Math.random()*w/32.0; }
if(y > h) { y = h-2.0; yi = Math.random()*(-h/32.0); }
if(y < 0) { y = 2.0; yi = Math.random()*h/32.0; }
if((rotate += 5)==360) { rotate = 0; }
if(scaleDirection==UP)
{
if((scale += 0.5) > 1.5) { scaleDirection = DOWN; }
}
else if(scaleDirection==DOWN)
{
if((scale -= 0.05) < 0.5) { scaleDirection = UP; }
}
}

public void paintComponent(Graphics g)
{
super.paintComponent(g);
Dimension d = getSize();
bi = new BufferedImage(d.width,d.height,BufferedImage.TYPE_INT_ARGB);
Graphics2D big = bi.createGraphics();
steps(d.width,d.height);
AffineTransform at = new AffineTransform();
at.setToIdentity();
at.translate(x,y);
at.rotate(Math.toRadians(rotate));
at.scale(scale,scale);
big.drawImage(image,at,this);
Graphics2D g2D = (Graphics2D) g;
g2D.drawImage(bi,0,0,null);
big.dispose();
}

public void start()
{
thread = new Thread(this);
thread.setPriority(Thread.MAX_PRIORITY);
thread.start();
}

public void stop()
```

```
     {
     if(thread != null) thread.interrupt();
     thread = null;
     }

    public void run()
    {
    Thread me = Thread.currentThread();
      while(thread == me) { repaint(); }
    thread  = null;
    }
}
```

10.3 AudioClip Class

JDK 1.2 provides us with the options to create and play AudioClips from both applets and applications. The clips can be any of the following audio file formats

```
AIFF
AU
WAV
MIDI (type 0 and type 1 file)
RMF
```

The `.au` audio clip format was developed by Sun. Windows uses the `.wav` format. For testing purposes, we can use the `.au` files in the Audio folders of the `JumpingBox` and `TicTacToe` demos. The

```
Interface java.applet.AudioClip
public abstract interface AudioClip
```

interface is a simple abstraction for playing a sound clip.

The sound engine can handle 8- and 16-bit audio data at virtually any sample rate. In JDK 1.2 audio files are rendered at a sample rate of 22 kHz in 16-bit stereo. If the hardware doesn't support 16-bit data or stereo playback, 8-bit or mono audio is output. The Java Sound engine minimizes the use of a system's CPU to process sound files. A full-featured Java Sound API is now available.

There are two ways to play sounds. The simplest uses the `play()` method, taking as its parameter the `URL` of the file

```
play(getCodeBase(),"ding.au");
```

This loads and plays the file immediately. The alternative is to set up an `AudioClip` object, load the file into there, and then play it later in the program

```
AudioClip music;
...
music = getAudioClip(getCodeBase(),"beep.au");
```

This gives us more control as we have access to three `AudioClip` methods

```
music.play();        // plays the clip once
music.loop();        // plays the clip repeatedly
music.stop();        // stops both loop or single play
```

The method

```
void loop()
```

starts playing this audio clip in a loop. The method

```
void play()
```

starts playing this audio clip. The method

```
void stop()
```

stops playing this audio clip.

```
// Sound1.java

import java.awt.*;
import java.applet.*;
import java.awt.event.*;

public class Sound1 extends Applet
{
   AudioClip music;
   Button quiet;

   public void init()
   {
   music = getAudioClip(getCodeBase(),"spacemusic.au");
   quiet = new Button("Stop that music");
   quiet.addActionListener(new B());
   add(quiet);
   music.loop();
   }

   class B implements ActionListener
   {
   public void actionPerformed(ActionEvent e)
   { music.stop(); }
   }

   static class WL extends WindowAdapter
   {
   public void windowClosing(WindowEvent e)
   { System.exit(0); }
   }
}
```

The classes for the Java Sound demo are contained in the `JavaSound.jar` file. To extract the contents of `JavaSound.jar` we run the command

```
jar xf JavaSound.jar
```

from the `JavaSound` directory. To run the Java Sound demo we enter

```
java -jar JavaSound.jar
```

The `JavaSound` demo consists of a set of demos housed in one GUI framework that uses `JTabbedPane`. We can access different groups of demos by clicking the tabs at the top of the pane. There are demo groups for Juke Box, Capture and Playback, Midi Synthesizer and Rhythm Groove Box. Using the Juke Box we can play sampled (`au`, `wav`, `aif`) and `midi` (`rmf`, `midi`) sound files. Using Capture and Playback we can record audio in different formats and then playback the recorded audio. The captured audio can be saved either as a `wav`, `au` or `aiff` file. We can load an audio file for streaming playback.

`MIDI` stands for musical instrument digitial interface. This connectivity standard enables us to hook together computers, musical instruments, and synthesizers to make and orchestrate digital sound. The term is used to describe the standard itself, the hardware that supports the standard, and the files that store information that the hardware can use. `MIDI` files are like digital sheet music. They contain instructions for musical notes, tempo, and instrumentation. They are widely used in game soundtracks and recording studios.

`AIFF` stands for Audio Interchange File Format. This audio file format was developed by Apple Computer for storing high-quality sampled audio and musical instrument information.

`WAV` is the Windows standard for waveform sound files. `WAV` files predictably have the extension `.wav`.

When a Sun Microsystems or other Unix computer make a noise, it does so in `AU` file format. Since the Internet is dominated by the Unix boxes, we find a lot of `AU` files there. Macintosh and PC browsers such as Netscape Navigator are able to play `AU` files, which have the extension `.au`.

Juke Box: A Juke Box for sampled (`au`, `wav`, `aif`) and `midi` (`rmf`, `midi`) sound files. Features duration progress, seek slider, pan and volume controls.

Capture and Playback: A Capture/Playback sample. Record audio in different formats and then playback the recorded audio. The captured audio can be saved either as a `wav`, `au` or `aif` file. We can also load an audio file for streaming playback.

Midi Synthesizer: Illustrates general `MIDI` melody instruments and `MIDI` controllers. A piano keyboard represents notes on and off. Features capture and playback of note on and off events.

Rhythm Groove Box: Illustrates how to build a track of short events. Features a tempo dial to increase or descrease the beats per minute.

In the following program we combine animation and sound.

```
// AniSoun.java

import java.awt.*;
import java.applet.*;

public class AniSoun extends Applet implements Runnable
{
   int i;
   Thread animator = null;
   Image[] pic = new Image[4];
   int current = 0;
   AudioClip music;

   public void init()
   {
   pic[0] = getImage(getCodeBase(),"figure1.jpg");
   pic[1] = getImage(getCodeBase(),"figure2.jpg");
   pic[2] = getImage(getCodeBase(),"figure3.jpg");
   pic[3] = getImage(getCodeBase(),"figure4.jpg");
   music = getAudioClip(getCodeBase(),"spacemusic.au");
   music.loop();
   }

   public void start()
   {
   if(animator == null)
   {
   animator = new Thread(this);
   animator.start();
   }
   }

   public void run()
   {

   while(animator != null)
```

```
   {
   i += 10;
   if(i > 400) { i = 0; }
   current++;
   if(current > 3) current = 0;
   repaint();

   try
   {
   Thread.sleep(100);
   } // end try

   catch(InterruptedException e) { }
   } // end while

   } // end run()

   public void paint(Graphics g)
   {
   g.drawImage(pic[current],i,100,this);
   }

}
```

The HTML file is

```
<HTML>
<APPLET CODE="AniSoun.class" width=275 height=135>
</APPLET>
</HTML>
```

Chapter 11

Networking

11.1 Introduction

A network client retrieves data from a server and shows it to a user. Web browsers are limited. They can only talk to certain kinds of servers (generally Web, FTP, Gopher, and perhaps main and news servers). They can only understand and display certain kinds of data (generally text, HTML, and a couple of standard image formats). A web browser cannot send SQL commands to a database server to perform interactive queries. Java programms can do this and a lot more. A Java program embedded in an HTML page (an applet) can give a Java-enabled web browser capabilities the browser did not have to begin with.

Java programs are flexible because Java is a general programming language. Java programs see network connections as streams of data which can be interpreted and responded to in any way that is necessary. Web browsers see only certain kinds of data streams and can interpret them only in certain ways. If a browser sees a data stream that it is not familiar with (for example, a binary response to an SQL query), its behavior is unpredictable.

Thus, a Java program can connect to a network time server to synchronize itself with an atomic clock. A web browser cannot. A Java program can connect to an Oracle database to ask for the salespeople for some data. A web browser cannot. Finally, a Java program can use the full power of a modern graphical user interface to show this data to the user.

Java includes built-in classes that connect to other Internet hosts, using the TCP and UDP protocols of the TCP/IP family. We tell Java what IP address and port we want, and Java handles the low-level details.

Once we have connected to a server, it is our responsibility to communicate properly with the remote server and interpret the data the server sends us. In almost all cases, packaging data to send to a server and unpacking the data we get back is harder than simply making the connection. Java includes classes that help us communicate with certain types of servers, notably web servers. It also includes classes to process some kinds of data, such as text, GIF images, and JPEG images. However, not all servers are HTTP servers, and not all data are GIFs. Therefore, Java lets us write protocol handlers that communicate with different kinds of servers and content handlers that communicate with different kinds of servers and content handers that understand and display different kinds of data. When the protocol or content handler is no longer needed, Java will automatically dispose of it so it does not waste memory or disk space.

A *network* is a collection of computers and other devices that can send data to and receive data from each other, more or less in real time. A network is normally connected by wires, and the bits of data are turned into electrons that move through the wires. However, wireless networks that transmit data through infrared light or microwaves are beginning to appear, and many long-distance transmissions are now carried over fiber optic cables that send visible light through glass filaments.

Each machine on a network is called a *node*. Most nodes are computers, but printers, routers, bridges, gateways and dumb terminals. Nodes that are fully functional computers are also called *hosts*. We will use the word node to refer to any device on the network, and host to mean that subset of devices which are general-purpose computers.

There are several different layer modes, each organized to fit the needs of a particular kind of network. Separating the network into layers lets us modify or even replace one layer without affecting the others, as long as the interfaces between the layers stay the same. Thus, applications talk only to the transport layer. The transport layer talks only to the application layer and the Internet layer. The Internet layer in turn talks only to the host-to-network layer and the transport layer, never directly to the application layer. The host-to-network layer moves the data across the wires, fiber-optic cables or other medium to the host-to-network layer on the remote system.

The *Internet* is the world's largest IP-based network. It is an amorphous group of computers in many different countries on all seven continents that talk to each other using the IP protocol. Each computer on the Internet has at least one unique IP address by which it can be identified. Most of them also have at least one name that maps to theat IP address. The Internet is not owned by anyone. It is simply a very large collection of computers that talk to each other in a standard way.

As described above IP addresses can exist in two forms:

1) The familiar DNS (Domain Name Service) form. The domain name of SUN is `sun.com`. My domain name is `whs.rau.ac.za`.

2) We can also use the dotted quad form, which is four numbers separated by dots, such as `207.25.71.25`.

The Internet is not the only IP-based network, but it is the largest one. Other IP networks are called *internets* with a little *i*; for example, a corporate IP network that is not connected to the Internet. *Intranet* is a current buzzword that loosely describes corporate practices of putting lots of data on internal web servers. Since web browsers use IP, most intranets do too (though some tunnel it through existing AppleTalk or IPX installations).

To make sure that hosts on different networks on the Internet can communicate with each other, a few rules that do not apply to purely internal internets need to be followed. The most important rules deal with the assignment of addresses to different organizations, companies, and individuals. If everyone randomly picked the Internet addresses they wanted, conflicts would arise when different computers showed up on the Internet with the same address. To avoid this problem, Internet addresses are assigned to different organizations by the Internet Network Information Center (InterNIC), generally acting through intermediaries called Internet Service Providers (ISPs). When an organization wants to set up an IP-based network, they are assigned a block of addresses by the InterNIC. Currently, these blocks are available in two sizes called Class B and Class C. A Class C address block specifies the first three bytes of the address, for example, `199.1.32`. This allows room for 254 individual addresses (from `1999.1.32.1` to `199.1.32.254`). A Class B address block only specifies the first two bytes of the addresses an organization may use, for instance, `167.1`. Thus a Class B address has room for roughly 65,000 different hosts (in block `167.1`, the hosts would have addresses from `167.1.0.1` to `167.1.255.254`).

IP Version 4 (IPv4) has three fundamental types of addresses: unicast, broadcast, and multicast. A *unicast address* is used to send a packet to a single destination. A *broadcast address* is used to send a datagram to an entire subnetwork. A *multicast address* is used to send a datagram to a set of hosts that can be on different subnetworks and that are configured as members of a multicast group. Multicast addresses are defined to be the IP addresses whose high-order four bits are `1110`, giving an address range from `224.0.0.0` through `239.255.255.255`. These addresses also are referred to as Class D addresses.

11.2 Addresses

Every network node has an IP (Internet protocoll) address: a series of bytes that uniquely identify it. Computers connected on the Internet are called hosts. Each host is identitfied by at least one unique 32-bit number called an Internet address, an IP address, or a host address. We write an IP as four unsigned bytes, each ranging from 0 to 255, with the most significant byte first. Bytes are separated by periods for human convenience. For example

```
152.106.50.60
```

Thus every computer on an IP network is identified by a four-byte number. In future IPv6 will use 16 byte (128 bit) addresses.

Addresses are assigned differently on different kinds of networks. AppleTalk addresses are chosen randomly at startup by each host. The host then checks to see if any other machine on the network is using the address. If another machine is using that address, the host randomly chooses another, checks to see if that address is already in use, and so on until it gets one that isn't being used. Ethernet addresses are attached to the physical Ethernet hardware. Manufacturers of Ethernet hardware use preassigned manufacturer codes to make sure there are no conflicts between the addresses in their hardware and the addresses of other manufacturers' hardware. Each manufacturer is responsible for making sure it doesn't ship two Ethernet cards with the same address. Internet addresses are normally assigned to a computer by the organization that is responsible for it. However, the addresses that an organization is allowed to choose for its computers are given out by and organisation called the InterNIC.

Addressing becomes important when we want to restrict access to our site. For instance, we may want to prevent a competing company from having access to our web site. In this case we would find out our competitor's address block and throw away all requests that come from there. More commonly, we might want to make sure that only people within our organization can access our internal web servers. In this case we would deny access to all requests except those that come from within our own address block.

There is no available block of addresses with a size between a Class B block and a Class C block. This has become a problem because there are many organizations with more than 254 and less than 65,000 computers, connected to the Internet. If each of these organizations is given a Class B address, a lot of IP addresses are wasted. This is a problem since the number of addresses is limited to about 4.2 billion. That sounds like a lot, but it gets crowded quickly when we waste fifty or sixty thousand addresses at a shot. The temporary solution is to assign multiple Class C addresses to these organizations, inconvenient though this is for packet filtering and routing.

Several address blocks and patterns are special. All Internet addresses beginning with 10. and 192. are deliberately unassigned. They can be used on internal networks, but no host using addresses in these blocks is allowed onto the global Internet. These non-routable addresses are useful for building private networks that can not be see from the rest of the Internet, or for building a large network when we have only been assigned a class C address block. Addresses beginning with 127 (most commonly 127.0.0.1) always mean the local loopback address. That is, these addresses always point to the local computer, no matter which computer we are running on. The hostname for this address is generally *localhost*. The address 0.0.0.0 always refers to the originating host, but may only be used as a source address, not a destination. Similarly, any address that begins with 0.0 is assumed to refer to a host on the same local network.

On some kinds of networks, nodes also have names that help human beings identify them. A particular name normally refers to exactly one address. The Domain Name System (DNS) was developed to translate hostnames that humans can remember such as

```
issc.rau.ac.za
```

into numeric Internet addresses

```
152.106.50.232
```

However, names are not locked to addresses. Names can change while addresses stay the same or addresses can change while the names stay the same. It is not uncommon for one address to have several names, and it is possible, though somewhat less common, for one name to refer to several different addresses.

When data is transmitted across the network, the packet's header includes the address of the machine for which the packet is intended (the destination address), and the address of the machine that sent the packet (the source address). Routers along the way choose the best route by which to send the packet by inpecting the destination address. The source address is included.

To get the IP address of the host machine we enter at the command prompt the command (Windows)

```
ipconfig
```

This provides us with the network configuration for the host machine. For example

```
Ethernet adapter
   IP address       152.106.50.60
   Subnet Mask      255.255.255.0
   Default Gateway  152.106.50.240
```

Under Linux the command is `ifconfig`. The Linux and Windows command `ping` sends echo request packets to a network host to see if it is accessible on the network. For example

```
ping 152.106.50.27
```

gives the reply

```
Ping statistics for 152.106.50.27
 Reply from 152.106.50.27: bytes = 32 time = 1ms TTL = 128
 Reply from 152.106.50.27: bytes = 32 time = 1ms TTL = 128
 Reply from 152.106.50.27: bytes = 32 time = 1ms TTL = 128
 Reply from 152.106.50.27: bytes = 32 time < 10 ms TTL = 128
 Packets: Sent 4, Received 4, Lost 0 (0% loss)
 Approximate round trip in milli-seconds:
 Minimum = 0ms, Maximum = 1ms, Average 0ms
```

The Linux command `finger` commmand looks up specified information about a user on the local system or remote systems. Users are specified by their username or first or last name and on a remote systems as `username@host`. With no users specified for the local system, all users logged in to the current system are listed. If a host with no username is provided in the form `host`, then a list of all users logged into the remote system is displayed. The `route` command displays or alters the IP routing table.

When Java programs access the network, they need to process both these numeric addresses and their corresponding hostnames. There are a series of methods for doing this in the `InetAddress` class. This class contains the methods

```
byte[] getAddress()
static InetAddress getByName(String name)
String getHostAddress()
String getHostName()
```

A URL (Uniform Resource Locator) consists of an address of a file on the Internet and a scheme for interpreting that file. The particular scheme of interest for HTML files is the HyperText Transfer Protocol (HTTP). For example to access the html file `diploma.html` on the Web site of the International School for Scientific Computing we enter at the address

```
http://zeus.rau.ac.za/diploma.html
```

We can access HTML files locally by replacing the `http:` scheme with the `file:` scheme and omitting the Internet address.

In the package `java.net` Java provides a number of classes useful for network programming. The classes are

```
DatagramPacket
DatagramSocket
HttpURLConnection
MulticastSocket
ServerSocket
Socket
URL
```

The `URL` class allows a URL instance to act as a reference for a resource available either locally or via the Web. The `URL` class has four constructors. The constructor

```
URL(String protocol,String host,int port,String file)
```

creates a `URL` object from the specified protocol, host, port number and file. The constructor

```
URL(String protocol,String host,String file)
```

creates an absolute `URL` from the specified protocol name, host name and file name. The constructor

```
URL(URL context,String spec)
```

creates a `URL` by parsing the specification `spec` within a specified context. The constructor

```
URL(String spec)
```

creates a `URL` object from the `String` representation.

Besides HTTP URLs we have File URLs, Gopher URLs, News URLs and others. Suppose there is a document called `mytext.txt` and it sits on an anonymous ftp server called `ftp.companyxxx.com` in the directory `/pub/files`. The URL for this file is then

```
ftp://ftp.companyxxx.com/pub/files/mytext.txt
```

Gopher URLs are a little more complicated than file URLs, since Gopher servers are a little tricker to deal with than FTP servers. To visit a particular gopher server say the gopher server on `gopher.companyxxx.com` we use this URL

```
gopher://gopher.companyxxx.com/
```

Some gopher servers may reside on unusual network ports on their host machine. The default gopher port number is 70. To point to a Usenet newsgroup (News URLs) for example `science.physics` the URL is simply

```
news:science.physics
```

11.3 Ports

Thus addresses would be all we needed if each computer did no more than one thing at a time. However, modern computers do many different things at once. Email needs to be separated from FTP requests, which need to be separated from web traffic. This is accomplished through *ports*. Each computer with an IP address has $2^{16} - 1 = 65535$ ports. These are provided in the computer's memory, and do not represent anything physical like a serial or parallel port. Each port is identified by a number between 1 and 65535. Each port can be allocated to a particular service.

A socket refers to a port on a specific location specified by the IP address. Internet communication requires an IP address and a port on the computer located at the specified IP address. There are 64kB ports available on each machine. Some of these are reserved for standard protocols as shown in Table 1. FTP is used to download files from the internet. Telnet supports remote login into a machine. SMTP is used to transport mail and POP to store mail and to deliver it to mail servers. HTTP is used for hypertext (Web) documents.

For example, the HTTP service, which is used by the Web, runs on port 80 - we say that a Web server listens on port 80 for incoming connections. SMTP or email servers run on port 25. When data is sent to a Web server on a particular machine at a particlar IP address, it is also sent to a particular port (usually port 80) on that machine. The receiver checks each packet for both the address and the port. If the address matches, the data is sent to the program interested in data sent to that port. This is how different types of traffic are sorted out.

Port numbers between 1 and 1023 are reserved for well-known services like finger, FTP, HTTP, and email. On UNIX systems, we must be "root" to receive data on one of these ports, but we may still send data to them. On Windows (including Windows NT) and the Mac, any user may use these ports without special privileges. Table 1 shows the well known ports for the protocols. These assignments are not absolutely guaranteed; in particular, Web servers often run on ports other than 80, either because multiple servers need to run on the same machine, or because the person who installed the server doesn't have the root privileges needed to run it on port 80.

We should of course avoid these common ports for our applications/applets except when specifically writing applications for these. It is generally recommended that we use ports in the range between 4000 and 8000 for our non-standard TCP/IP applications.

On UNIX machines, a fairly complete listing of assigned ports is stored in the file `/etc/services`.

Table 1: Port Assignments

Protocol	Port	Encoding	Purpose
echo	7	tcp/udp	Echo is a test protocol used to verify that two machines are able to connect by having one echo back the other's input.
discard	9	tcp/udp	Discard is a less useful test protocol that ignores all data received by the server.
daytime	13	tcp/udp	Daytime provides an ASCII representation of the current time on the server.
ftp-data	20	tcp	FTP uses two well-known ports. This port is used to transfer files.
ftp	21	tcp	This port is used to send ftp commands like `put` and `get`.
telnet	23	tcp	Telnet is a protocol used for interactive, remote command-line sessions.
smtp	25	tcp	The "Simple Mail Transfer Protocol" is used to send email between machines.
time	37	tcp/udp	A time server returns the number of seconds that have elapsed on the host machine since midnight, January 1, 19000, as a four-byte signed, big-endian integer.
whois	43	tcp	Whois is a simple directory service for Internet network administrators.
finger	79	tcp	Finger gets information about a user or users.
http	80	tcp	HyperText Transfer Protocol is the underlying protocol of the World Wide Web.
pop3	110	tcp	Post Office Protocol Version 3 is a protocol for the transfer of accumulated email from the host to sporadically connected clients.
nntp	119	tcp	This is the Usenet news transfer, more formally known as the "Network News Transfer Protocol".
RMI Registry	1099	tcp	The Remote Method Invocation (RMI) Registry is a registry service for Java remote objects.
Jeeves	8080	tcp	Jeeves *The Java Server API and Servlets,* is a web server from Sun that runs on port 8080 by default, not port 80. This non-standard port is used because Jeeves is still experimental software.

11.4 Examples

After we compiled the following program using `javac Net1.java` we enter at the command line:

```
java Net1 whs.rau.ac.za
```

The output is:

```
Host name: whs.rau.ac.za
IP address: 152.106.50.60
```

```
// Net1.java

import java.io.*;
import java.net.*;

public class Net1
{
   public static void main(String[] args)
   {
   try
   {
   if(args.length != 1)
   {
   System.err.println("Usage: Net1 MachineName");
   return;
   }
   InetAddress host = InetAddress.getByName(args[0]);
   String hostName = host.getHostName();
   byte ipAddress[] = host.getAddress();

   System.out.println("Hostname = " + hostName);
   System.out.print("IP address: " );

   for(int i=0;i<ipAddress.length;++i)
   System.out.print((ipAddress[i]+256)%256+".");
   System.out.println();
   }
   catch(UnknownHostException ex)
   {
   System.out.println("Unkown host");
   return;
   }
   }
}
```

The following program finds the IP address of the local machine. If I run this program on my machine I find: `My address is: 152.106.50.60`

```
// Net2.java

import java.net.*;

public class Net2
{
   public static void main(String args[])
   {

   try
   {
   InetAddress thisComputer = InetAddress.getLocalHost();
   byte[] address = thisComputer.getAddress();
   System.out.println("My address is: ");
   for(int i=0; i<address.length; i++)
   {
   int unsignedByte = address[i] < 0 ? address[i] + 256 : address[i];
   System.out.print(unsignedByte + ".");
   }
   System.out.println();
   } // end try block

   catch(UnknownHostException e)
   {
   System.out.println("Sorry. I do not know my own address.");
   } // end catch block
   } // end main
}
```

In the following program we get the domain name for a given IP address in the present case 152.106.50.232. If we run this program we find the output

```
issc.rau.ac.za/152.106.50.232
```

```
// Net3.java

// Prints the address of 152.106.50.232

import java.net.*;

public class Net3
{
   public static void main(String args[])
   {

   try
   {
   InetAddress address = InetAddress.getByName("152.106.50.232");
   System.out.println(address);
   }  // end try block

   catch(UnknownHostException e)
   {
   System.out.println("Could not find 152.106.50.232");
   }  // end catch block

   }
}
```

In the following program we find the IP address of a given domain name. When we run this program we find

```
issc.rau.ac.za/152.106.50.232
```

```
// Net4.java

// Prints all addresses of issc.rau.ac.za

import java.net.*;

public class Net4
{
   public static void main(String args[])
   {

   try
   {
   InetAddress[] addresses =
   InetAddress.getAllByName("issc.rau.ac.za");
   for(int i=0; i<addresses.length; i++)
   {
   System.out.println(addresses[i]);
   }
   } // end try block

   catch(UnknownHostException e)
   {
   System.out.println("Could not find issc.rau.ac.za");
   } // end catch block

   }
}
```

11.5 URL Class

The URL class provides us with the following methods

```
public String getProtocol()
public String getHost()
public int getPort()
public String getFile()
public String getRef()
```

The getProtocol() method returns a String containing the protocol portion of the URL, for example http or file. The getRef() method returns the named anchor of the URL. If the URL does not have an anchor, the method returns null. The following program shows how these methods are applied.

```
// URLmethods.java

import java.net.*;

public class URLmethods
{
   public static void main(String args[])
   {
   int i;
   for(i = 0; i < args.length; i++)
   {
   try
   {
   URL u = new URL(args[0]);
   System.out.println("The URL is " + u);
   System.out.println("The protocol part is " + u.getProtocol());
   System.out.println("The host part is " + u.getHost());
   System.out.println("The port part is " + u.getPort());
   System.out.println("The file part is " + u.getFile());
   System.out.println("The ref part is " + u.getRef());
   } // end try
   catch(MalformedURLException e)
   {
   System.err.println(args[0] + "is not a URL I understand.");
   } // end catch
   } // end for
   } // end main
}  // end URLmethods
```

Running the program with the command

```
java URLmethods http://issc.rau.ac.za/symbolic/symbolic.html
```

we get the output

```
The URL is http://issc.rau.ac.za/symbolic/symbolic.htm
The protocol part is http
The host part is issc.rau.ac.za
The port part is -1
The file part is /symbolic/symbolic.html
The ref part is null
```

With the following program we can download an HTML file using the `URL` class. After compiling `Viewsource.java`, i.e. we generated the file

```
Viewsource.class
```

then at the command line type:

```
java Viewsource http://www.amd.com/products/lpd/lpd.html
```

this will download the HTML file `lpd.html` from AMD (Advanced Micro Devices). If we want redirect to output to the file `mylpd.htm` on our hard disc type:

```
java Viewsource http://www.amd.com/products/lpd/lpd.html > mylpd.htm
```

```
// Viewsource.java

import java.net.*;
import java.io.*;

public class Viewsource
{
   public static void main(String args[])
   {
   String thisLine;
   URL u;               // URL is a Uniform Resource Locator
                        // A pointer to a resource on the WWW.
   if(args.length > 0)
   {
   try
   {
   u = new URL(args[0]);

   try
   {
   BufferedReader theHTML =
```

```
   new BufferedReader(new InputStreamReader(u.openStream()));

   try
   {
   while((thisLine = theHTML.readLine()) != null)
   {
   System.out.println(thisLine);
   } // while loop ends here
   } // end try

   catch(Exception e)   {   System.err.println(e);   }
   } // end try

   catch(Exception e)   {   System.err.println(e);   }
   } // end try

   catch(MalformedURLException e)
   {
   System.err.println(args[0] + " is not a parseable URL");
   System.err.println(e);
   } // end catch
   } // end if

   } // end main
}
```

Under Microsoft Internet Explorer we can directly get the HTML file from the server. After we loaded the HTML file in our Web browser we click at `View`. Then we click at `Source` and we are provided with the HTML file.

11.6 Socket Class

A *socket* is an endpoint for communication between two machines. Sockets are an innovation of Berkeley UNIX. The allow the programmer to treat a network connection as another stream that bytes can be written onto or read from. Java provides a `Socket` class. The class implements client sockets. The actual work of the socket is performed by an instance of the `SocketImpl` class. The constructor

```
Socket()
```

creates an unconnected socket, with the system-default type of `SocketImpl`. The constructor

```
Socket(InetAddress address,int port)
```

creates a stream socket and connects it to the specified IP address.

The method

```
int getLocalPort()
```

returns the local port to which to socket is bound. The method

```
int getPort()
```

returns the remote port to which the socket is connected.

If we run the following program with the command

```
java MySocket issc.rau.ac.za
```

we obtain the output

```
The is a server on port 80 of issc.rau.ac.za
The is a server on port 135 of issc.rau.ac.za
The is a server on port 139 of issc.rau.ac.za
The is a server on port 443 of issc.rau.ac.za
```

```
// MySocket.java

import java.net.*;
import java.io.*;

public class MySocket
{
   public static void main(String[] args)
   {
   Socket s;
   String host = "localhost";

   if(args.length > 0)
   {
   host = args[0];
   }

   int i = 0;
   while(i < 1024)
   {
   try
   {
   s = new Socket(host,i);
   System.out.println("There is a server on port " + i + " of " + host);
   } // end try
   catch(UnknownHostException e)
   {
   System.err.println(e);
   }
   catch(IOException e)
   {  }
   i++;
   } // end while

   } // end main
}
```

11.7 Client-Server Application

As described above a socket is an endpoint of communication. A socket in use usually has an address bound to it. The nature of the address depends on the communication domain of the socket. There are several socket types, which represent classes of services. Stream sockets provide reliable, duplex, sequenced data streams. No data is lost or duplicated in delivery, and there are no record boundaries. This type is supported in the Internet domain by the TCP protocol. In the Unix domain, pipes are implemented as a pair of communicating stream sockets. The Unix input/output (I/O) system follows a paradigm usually referred to as `Open-Read-Write-Close`. Before a user process can perform I/O operations, it calls `Open` to specify and obtain permissions for the file or device to be used. Once an object has been opened, the user process makes one or more calls to `Read` or `Write` data. `Read` reads data from the object and transfers it to the user process, while `Write` transfers data from the user process to the object. After all transfer operations are complete, the user process calls `Close` to inform the operating system that it has finished using that object. When facilities for InterProcess Communication (IPC) and networking were added to Unix, the idea was to make the interface to IPC similar to that of file I/O. In Unix, a process has a set of I/O descriptors that one reads from and writes to. These descriptors may refer to files, devices, or communication channels (sockets). The lifetime of a descriptor is made up of three phases: creation (open socket), reading and writing (receive and send to socket), and destruction (close socket). The IPC interface in BSD-like versions of Unix is implemented as a layer over the network TCP and UDP protocols. Message destinations are specified as socket addresses; each socket address is a communication identifier that consists of a port number and an Internet address.

The IPC operations are based on socket pairs, one belonging to a communication process. IPC is done by exchanging some data through transmitting that data in a message between a socket in one process and another socket in another process. When messages are sent, the messages are queued at the sending socket until the underlying network protocol has transmitted them. When they arrive, the messages are queued at the receiving socket until the receiving process makes the necessary calls to receive them.

TCP/IP and UDP/IP communications

There are two communication protocols that one can use for socket programming: datagram communication and stream communication.

The datagram communication protocol, known as UDP (user datagram protocol), is a connectionless protocol, meaning that each time we send datagrams, we also need to send the local socket descriptor and the receiving socket's address. As we can tell, additional data must be sent each time a communication is made.

Stream communication: The stream communication protocol is known as TCP (transfer control protocol). Unlike UDP, TCP is a connection-oriented protocol. In order to do communication over the TCP protocol, a connection must first be established between the pair of sockets. While one of the sockets listens for a connection request (server), the other asks for a connection (client). Once two sockets have been connected, they can be used to transmit data in both (or either one of the) directions.

What protocol we should use depends on the client/server application we are writing.

In UDP every time we send a datagram, we have to send the local descriptor and the socket address of the receiving socket along with it. Since TCP is a connection-oriented protocol, on the other hand, a connection must be established before communications between the pair of sockets start. So there is a connection setup time in TCP. In UDP, there is a size limit of 64 kilobytes on datagrams we can send to a specified location, while in TCP there is no limit. Once a connection is established, the pair of sockets behaves like streams: All available data are read immediately in the same order in which they are received. UDP is an unreliable protocol – there is no guarantee that the datagrams we have sent will be received in the same order by the receiving socket. On the other hand, TCP is a reliable protocol; it is guaranteed that the packets we send will be received in the order in which they were sent. In short, TCP is useful for implementing network services – such as remote login (rlogin, telnet) and file transfer (FTP) – which require data of indefinite length to be transferred. UDP is less complex and incurs fewer overheads. It is often used in implementing client/server applications in distributed systems built over local area networks.

Programming Sockets in Java

We give an examples of how to write client and server applications. We consider the TCP/IP protocol only since it is more widely used than UDP/IP. All the classes related to sockets are in the `java.net` package, so make sure to import that package.

If we are programming a client, then we would open a socket like this:

```
Socket MyClient;
MyClient = new Socket("Machine name",PortNumber);
```

Where Machine name is the machine we are trying to open a connection to, and PortNumber is the port (a number) on which the server we are trying to connect to is running. When selecting a port number, we should note that port numbers between 0 and 1023 are reserved for privileged users (that is, super user or root). These port numbers are reserved for standard services, such as email, FTP, and HTTP. When selecting a port number for our server, select one that is greater than 1023. Obviously we should also include exception handling. The above can be written as:

```
Socket MyClient;
try {
  MyClient = new Socket("Machine name",PortNumber);
}
catch(IOException e) {
System.out.println(e);
}
```

If we are programming a server, then this is how we open a socket:

```
ServerSocket MyService;
try {
 MyServerice = new ServerSocket(PortNumber);

}
catch(IOException e) {
 System.out.println(e);
}
```

When implementing a server we need to create a socket object from the `ServerSocket` in order to listen for and accept connections from clients.

```
Socket clientSocket = null;
try {
 serviceSocket = MyService.accept();
}
catch(IOException e) {
 System.out.println(e);
}
```

On the client side, we can use the `DataInputStream` class to create an input stream to receive response from the server

```
DataInputStream input;
try {
 input = new DataInputStream(MyClient.getInputStream());
}
catch(IOException e) {
 System.out.println(e);
}
```

The class `DataInputStream` allows us to read lines of text and Java primitive data types in a portable way. It has methods such as

```
read, readChar, readInt, readDouble, readLine
```

Use whichever function we think suits our needs depending on the type of data that we receive from the server. On the server side, we can use `DataInputStream` to receive input from the client

```
DataInputStream input;
try {
   input = new DataInputStream(serviceSocket.getInputStream());
}
catch(IOException e) {
 System.out.println(e);
}
```

On the client side, we can create an output stream to send information to the server socket using the class `PrintStream` or `DataOutputStream` of `java.io`

```
PrintStream output;
try {
 output = new PrintStream(MyClient.getOutputStream());
}
catch(IOException e) {
 System.out.println(e);
}
```

The class `PrintStream` has methods for displaying textual representation of Java primitive data types. Its `write()` and `println()` methods are important here. Also, we may want to use the `DataOutputStream`

```
DataOutputStream output;
try {
 output = new DataOutputStream(MyClient.getOutputStream());
}
catch(IOException e) {
 System.out.println(e);
}
```

The class `DataOutputStream` allows us to write Java primitive data types; many of its methods write a single Java primitive type to the output stream. The method `writeBytes` is a useful one. On the server side, we can use the class `PrintStream` to send information to the client.

```
PrintStream output;
try {
 output = new PrintStream(serviceSocket.getOutputStream());
}
catch(IOException e) {
 System.out.println(e);
}
```

We should always close the output and input stream before we close the socket. On the client side

```
try {
 output.close();
 input.close();
 MyClient.close();
}
catch(IOException e) {
 System.out.println(e);
}
```

On the server side

```
try {
 output.close();
 input.close();
 serviceSocket.close();
 MyService.close();
}
catch(IOException e) {
 System.out.println(e);
}
```

The following two programs show an application. We first compile the two programs

```
javac MyClient.java
javac MyServer.java
```

Then we start the server with `java MyServer`. Under Unix (Linux) we start the server as background process

```
java MyServer &
```

Then we start the client with

```
java MyClient
```

In the following example we run the two programs on the same machine. Under Windows we open first a window for the server process and start it at the prompt with `java MyServer`. Then we open another window and start at the prompt the client `java MyClient`.

For the client we have

```
// MyClient.java

import java.io.*;
import java.net.*;

public class MyClient
{
   public static void main(String[] args) throws IOException
   {
   int c;

   Socket connection;
   InputStream fromServer;

   // can also be written as
   // connection = new Socket("127.0.0.1",5321);
   connection = new Socket("localhost",5321);

   fromServer = connection.getInputStream();

   while((c = fromServer.read()) != -1)
   {
   System.out.print((char) c);
   }

   fromServer.close();
   connection.close();
   }
}
```

For the server we have

```
// MyServer.java

import java.io.*;
import java.net.*;

public class MyServer
{
   public static void main(String[] args) throws IOException
   {
   Socket connection;
   ServerSocket servSocket = (ServerSocket) null;

   String sendString = "Good Morning Egoli\n";
   int sLength;
   OutputStream toClient;

   try
   {
   servSocket = new ServerSocket(5321);
   } catch(IOException e) { }

   while(true)
   {
   try
   {
   connection = servSocket.accept();

   toClient = connection.getOutputStream();
   sLength = sendString.length();

   for(int i=0;i<sLength;i++)
   {
   toClient.write((int) sendString.charAt(i));
   }

   toClient.close();
   connection.close();
   } catch(IOException e) { }
   } // end while

   } // end main
} // end class
```

The following programs show an application of UDP (User Datagram Protocol). On the receiver side (with the IP address 152.106.50.69) we compile and run `udp_r.java`. Then on the sender side we compile `udp_s.java` and run it with, for example,

```
java udp_s 152.106.50.69 "Good Morning Egoli"
```

```
// upd_s.java
import java.net.*;
import java.io.*;

public class udp_s {
   static final int pNumber = 1234;
   public static void main(String args[]) throws Exception
   {
   InetAddress addr = InetAddress.getByName(args[0]);
   DatagramSocket s = new DatagramSocket();
   System.out.println( "Destination is:  " + addr.getHostName() +
       " (port " + pNumber + ")" );
   System.out.println( "Message is:      \"" + args[1]);
   DatagramPacket packet =
     new DatagramPacket(args[1].getBytes(),args[1].length(),addr,pNumber);
   s.send(packet);
   System.out.println("Datagram sent.");
   }
}
```

```
// udp_r.java
import java.net.*;
import java.io.*;

public class udp_r {
    static final int pNumber = 1234;
    static DatagramSocket socket;
    public static void main(String args[]) throws Exception
    {
    socket = new DatagramSocket(pNumber);
    while(true) {
      byte buffer[] = new byte[65535];
      DatagramPacket packet = new DatagramPacket(buffer,buffer.length);
      socket.receive(packet);
      String s = new String(packet.getData());
      s = s.substring(0,packet.getLength());
      System.out.println("Datagram received: \"" + s + "\"");
     }
     }
}
```

11.8 Remote Method Invocation

Java Remote Method Invocation (RMI) allows a client application to call methods on an object residing on a remote server as if it were a local object. An application becomes server by implementing a remote interface and extending a server reference class. The remote interface declares the methods that are available on that object to clients possesing a reference to that object. Each machine with an RMI server has a Registry residing on that maps the name of an object to an object reference. When a client desires to connect to a server it queries the Registry on the correct machine for an object with a certain name and uses the returned reference to make method calls on the remote object. Stubs generated by the `rmic` (Remote Method Invokation Compiler) serve as a client `proxy` to the remote object. Marshaling means converting a method argument from its native binary representation into some language-independent format, and then converting this generic representation into a binary format that is appropiate to the called method.

The source files are

- `AppRMI.java`, on the *client* site and the *server* site, describes the interface used by the server and client to cooperate. The precise interaction is described by the methods of the `AppRMI` interface.
- `AppRMIClient.java` on the *client* site is the source for the client program.
- `AppRMIServer.java` on the *server* site is the source for the server program.
- `AppRMIImpl.java` on the *server* site implements the methods which a client program can execute. In other words it implements the `AppRMI` interface.
- `java.policy` on the *server* and *client* sites, describes the security restrictions placed on the JAVA programs.

The conventions we use are

- `CLIENT` the directory on the client site local filesystem where compiled classes will be placed. For example under Windows it could be `c:\client\rmi` and under Linux `~/client/rmi`.
- `SERVER` the directory on the server site local filesystem where compiled classes will be placed. For example under Windows it could be `c:\serverrmi` and under Linux `~/serverrmi`.
- `WEBDIR` the directory on the server site local filesystem where compiled classes, which need to be accessed via `http`, will be placed. For example under Windows it could be `c:\webdocs\rmi` and under Linux `~/public_html/`.
- `HOST` the server hostname, for example

```
issc.rau.ac.za
```

- `CODEBASE` the `http` URL specifying where the necessary server classes can be obtained by the client, for example

  ```
  http://issc.rau.ac.za/rmi/
  ```

  The trailing `/` is required.

For `javac` and `rmic` compilers the command line option

```
-d directory
```

places generated `.class` files in the specified directory. For example

```
javac -d directory someclass.java
```

creates the file

```
directory\someclass.class
```

in Windows and

```
directory/someclass.class
```

in UNIX and Linux.

For `javac` and `java` the `-classpath path` option specifies where to find `.class` files. The option overrides the `CLASSPATH` environment variable. To run the above example program with the classpath specified by the environment variable `CLASSPATH` we would enter on the command line

```
java -classpath directory;%CLASSPATH% someclass
```

in Windows and

```
java -classpath directory:$CLASSPATH someclass
```

in UNIX and Linux.

The program `rmiregistry` (`rmiregistry.exe`), provided with the JDK (and located in the `jdk1.4/bin` directory), is used to provide an interface between clients and servers. Using the RMI registry a server can register services (make them available to clients) in the form of implementations of interfaces. The client can use the registry to determine if a service is available, and obtain a reference to an `Object` which can be cast into the appropriate interface, which allows the service to be used.

The program `rmic` (`rmic.exe`) or RMI compiler program (located in the `jdk1.4/bin` directory) is used to generate the stub classes used by the `rmiregistry` program. The stub classes facilitate the intercation between client and server programs. For `rmic` the option `-v1.2` is used to generate only JAVA 1.2 compatible stub class files. For example

```
rmic -d directory someclass
```

creates the files

```
directory\someclass_Skel.class
directory\someclass_Stub.class
```

in Windows and

```
directory/someclass_Skel.class
directory/someclass_Stub.class
```

in UNIX and Linux. The command

```
rmic -v1.2 -d directory someclass
```

creates the file

```
directory\someclass_Stub.class
```

in Windows and

```
directory/someclass_Stub.class
```

in UNIX and Linux.

For `java` (execute file) the option `-Dproperty=value` sets the system property `property` to `value`. The system properties we use are

- `java.rmi.server.codebase` which is the URL specifying where classes can be obtained by clients.
- `java.rmi.server.hostname` which is the hostname identifying the server machine.
- `java.security.policy` which is a filename in the local filesystem which provides restrictions to certain JAVA functionality.

For example the command (one line)

```
java -Djava.rmi.server.codebase=http://issc.rau.ac.za/rmi/
     -Djava.rmi.server.hostname=issc.rau.ac.za
     -Djava.security.policy=java.policy
     someclass
```

runs the program `someclass` with the security policy specified by `java.policy` in the current directory, the RMI server codebase at

`http://issc.rau.ac.za/rmi/`

and the RMI server hostname

`issc.rau.ac.za`

Compiling the Programs

Next we have to compile the programs. On the **server site** we do the following. We use the following sequence of commands on the command line

```
javac -d SERVER AppRMI.java
javac -d WEBDIR AppRMI.java
javac -classpath SERVER -d SERVER AppRMIImpl.java
javac -classpath SERVER -d SERVER AppRMIServer.java
rmic -v1.2 -d WEBDIR AppRMIImpl
```

where `WEBDIR` is (in our case) for Windows `c:\webdocs\rmi`. In Linux we could use `~/public_html/rmi`. The first line makes the interface `AppRMI` available to the server program and implementation class. The second line makes the interface `AppRMI` available to client programs and the `rmiregistry` program. The third and fourth lines compile the server program and implementation class. The fifth line uses the `rmic` compiler to generate a stub class for remote access to the `AppRMIImpl` class. The stub class is used by the `rmiregistry` program and the client program.

On the **client site** we do the following. We use the following sequence of commands on the command line

```
javac -d CLIENT AppRMI.java
javac -classpath CLIENT -d CLIENT AppRMIClient.java
```

The first line makes the interface `AppRMI` available to the client program. The second line compiles the client program.

Running the Programs

Next we run the programs. To start the **server** we do the following. First we must start the `rmiregistry` program (`rmiregistry.exe`). This must be done without a classpath, and in a directory where none of the server `.class` files exist. To unset the classpath we can use

```
set CLASSPATH=
```

in Windows and

```
unsetenv CLASSPATH
```

in the `sh` and `csh` shells and

```
export -n CLASSPATH
```

in the `bash` shell.

We start the `rmiregistry` program by entering

```
start rmiregistry
```

in Windows and

```
rmiregistry &
```

in UNIX and Linux. The ampersand (`&`) in Unix and Linux indicates that the program is to be run in the background, i.e. the current shell continues to run immediately after the command and more commands can be issued.

Now we can set the `CLASSPATH` again if necessary. Lastly we start the server program with the command (**single line**)

```
java -classpath SERVER;WEBDIR
     -Djava.rmi.server.codebase=CODEBASE
     -Djava.rmi.server.hostname=HOST
     -Djava.security.policy=java.policy
     AppRMIServer
```

where in our case `CODEBASE` would be `http://issc.rau.ac.za`, `HOST` would be `issc.rau.ac.za`, `WEDDIR` is `c:\webdocs\rmi` and `SERVER` is the directory on the server site local file system.

The option

```
-classpath SERVER;WEBDIR
```

is in the Windows path format, for UNIX and Linux we would use

```
-classpath SERVER:WEBDIR
```

instead. Next we have to start the client. We start the client program using (**single line**)

```
java -classpath CLIENT -Djava.security.policy=java.policy
     AppRMIClient HOST
```

where HOST would be issc.rau.ac.za.

Example

Let us give an example. We use all the previous instructions for building and running the programs. In this case the filenames begin with Lotto instead of App since we created a Lottery number generating system. The java.policy file used is given below

```
grant {
permission java.net.SocketPermission "*:1024-65535", "connect,accept";
permission java.net.SocketPermission "*:80", "connect";
};
```

which provides enough network access to retrieve the necessary .class files via http (port 80), and to use the RMI registry (usually on port 1099).

The interface LottoRMI.java specifies that the only method we can use to communicate with the server is the findnumbers() method. The file LottoRMI.java is on the server and client site.

```
// LottoRMI.java

public interface LottoRMI extends java.rmi.Remote
{
 int[] findnumbers() throws java.rmi.RemoteException;
}
```

The client program must create a SecurityManager to allow access to the server codebase via http. This is achieved by creating a RMISecurityManager() which implements the security restrictions defined in the file specified by java.security.policy. The method Naming.lookup(String) is used to refer to a class on a RMI server. The string argument is an URL of the form

```
 rmi://host:port/name
```

where host is the name of the RMI server machine, port is a port number and name specifies the class the client wishes to refer to. The rmi: part of the URL is not required, if the :port part of the URL not given, the port is assumed to be 1099. The method Naming.lookup() return type is Remote, which must be cast into the appropriate interface (LottoRMI in this case). This service is provided by the rmiregistry program. The file LottoRMIClient.java is on the client site.

```
// LottoRMIClient.java

import java.rmi.*;
```

```
import java.rmi.registry.*;
import java.rmi.server.*;

public class LottoRMIClient
{
 public static void main(String args[])
 {
 System.setSecurityManager(new RMISecurityManager());
 try
 {
 LottoRMI lotto=(LottoRMI)Naming.lookup("//"+args[0]+"/lottormi");
 int[] numbers=lotto.findnumbers();
 System.out.println("Your lucky numbers are: ");
 for(int i=0;i<numbers.length;i++)
 {
 System.out.println(numbers[i]);
 }
 }
 catch(Exception e)
 {
 System.err.println("System Exception"+e);
 e.printStackTrace();
 }
 System.exit(0);
 }
}
```

The server program must create a `SecurityManager` in the same way as the client. The server only makes the implementation class `LottoRMIImpl` available to client programs. The file `LottoRMIServer.java` is on the server site.

```
// LottoRMIServer.java

import java.rmi.*;
import java.rmi.server.*;

public class LottoRMIServer
{
 public static void main(String args[])
 {
  System.setSecurityManager(new RMISecurityManager());
  try
  {
  LottoRMIImpl test=new LottoRMIImpl("lottormi");
```

```
  System.out.println("LottoRMI Server ready.");
  }
  catch(Exception e)
  {
  System.out.println("Expception: "+e.getMessage());
  e.printStackTrace();
  }
 }
}
```

The class `LottoRMIImpl` provides the implemetations of the methods which can be remotely invoked by the client program. The method `Naming.rebind(String)` associates a `String` with the implementation class. The `String` parameter is the `name` part of the URL specified by the client program. This service is provided by the `rmiregistry` program. The file `LottoRMIImpl.java` is on the server site.

```
// LottoRMIImpl.java

import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class LottoRMIImpl extends UnicastRemoteObject
                          implements LottoRMI
{
 private int numbers=6;
 private int maximum=50;

 public LottoRMIImpl(String name) throws RemoteException
 {
  super();
  try
  {
   Naming.rebind(name,this);
  }
  catch(Exception e)
  {
   System.out.println("Exception: "+e.getMessage());
   e.printStackTrace();
  }
 }

 public int[] findnumbers() throws RemoteException
 {
  int[] array = new int[numbers];
```

```
  for(int i=0;i<numbers;i++)
  {
   array[i]=(int)(maximum*Math.random());
   for(int j=0;j<i;j++)
    { if(array[i]==array[j]) { j=i; i--; } }
  }
  return array;
 }
}
```

11.9 SocketChannel and ServerSocketChannel

Instead of using RMI for a network task in many cases the same task can be accomplished more simply and with less overhead using the

`SocketChannel` `ServerSocketChannel`

classes in the `java.nio` package. In our example we send an array of 4 `double` and the length of the array to the Server. The Server calculates the arithmetic, geometric and harmonic mean of the 4 numbers. Then it returns these three numbers and the length of the original array. Enough memory must be allocated for the `ByteBuffer`. We recall that the size of `int` is 4 bytes and the size of `double` is 8 bytes.

The class `Buffer` is a container of a specific primitive data type (excluding `boolean`). A buffer is a linear, finite sequence of elements it contains. The capacity of a buffer is never negative and never changes. The method

`Buffer flip()`

flips this buffer. The limit is set to the current position and then the position is set to zero. The method

`DoubleBuffer put(int index,double d)`

in class `DoubleBuffer` writes the given `double` into this buffer at the given index. The method

`double get(int index)`

in class `DoubleBuffer` reads the `double` at the given index.

```
// MeanClient.java

import java.nio.channels.SocketChannel;
import java.nio.ByteBuffer;
import java.nio.DoubleBuffer;
import java.nio.IntBuffer;
import java.io.IOException;
import java.net.InetSocketAddress;

public class MeanClient
{
   static int nlength = 4;
   static double[] narray = new double[nlength];
   private SocketChannel channel;
   private ByteBuffer buffer =
```

```
          ByteBuffer.allocate(48);
   private DoubleBuffer doubleBuffer =
              buffer.asDoubleBuffer();
   private IntBuffer intBuffer = buffer.asIntBuffer();

   public void getMean(int nlength,double narray[])
   {
   try
   {
   channel = connect();
   sendMeanRequest(nlength,narray);
   receiveResponse();
   }
   catch(IOException e)
   {
   e.printStackTrace();
   }
   finally
   {
   if(channel != null)
   {
   try
   {
   channel.close();
   }
   catch(IOException e)
   {
   e.printStackTrace();
   }
   }
   }
}

   private SocketChannel connect() throws IOException
   {
   InetSocketAddress socketAddress =
//       new InetSocketAddress("issc.rau.ac.za",9099);
          new InetSocketAddress("localhost",9099);
   return SocketChannel.open(socketAddress);
   }

   private void sendMeanRequest(int nlength,double[] narray)
          throws IOException
   {
   buffer.clear();
```

```
   intBuffer.put(0,nlength);
   for(int i=1; i<=nlength; i++)
   {
   doubleBuffer.put(i,narray[i-1]);
   }
   channel.write(buffer);
   }

   private void receiveResponse() throws IOException
   {
   buffer.clear();
   channel.read(buffer);
   int nlength = intBuffer.get(0);
   double arithmean = doubleBuffer.get(1);
   double harmmean = doubleBuffer.get(2);
   double geommean = doubleBuffer.get(3);
   System.out.println("size of array = " + nlength);
   System.out.println("arithmetic mean = " + arithmean);
   System.out.println("harmonic mean = " + harmmean);
   System.out.println("geometric mean = " + geommean);
   }

   public static void main(String[] args)
   {
   narray[0] = 3.4;  narray[1] = 7.6;
   narray[2] = 17.1; narray[3] = 20.4;
   new MeanClient().getMean(nlength,narray);
   }
}


// MeanServer.java

import java.nio.channels.SocketChannel;
import java.nio.channels.ServerSocketChannel;
import java.nio.ByteBuffer;
import java.nio.IntBuffer;
import java.nio.DoubleBuffer;
import java.io.IOException;
import java.net.InetSocketAddress;

public class MeanServer
{
   ByteBuffer buffer = ByteBuffer.allocate(48);
   DoubleBuffer doubleBuffer = buffer.asDoubleBuffer();
```

```
   IntBuffer intBuffer = buffer.asIntBuffer();
   ServerSocketChannel channel = null;
   SocketChannel sc = null;

   public void start()
   {
   try
   {
   openChannel();
   waitForConnection();
   }
   catch(IOException e)
   {
   e.printStackTrace();
   }
   }

private void openChannel() throws IOException
{
   channel = ServerSocketChannel.open();
   channel.socket().bind(new InetSocketAddress(9099));
   while(!channel.isOpen())
   {  }
   System.out.println("Channel is open...");
}

private void waitForConnection() throws IOException
{
   while(true)
   {
   sc = channel.accept();
   if(sc != null)
   {
   System.out.println("A connection is added...");
   processRequest();
   sc.close();
   }
   }
}

private void processRequest() throws IOException
{
   buffer.clear();
   sc.read(buffer);
   int nlength = intBuffer.get(0);
```

```
   System.out.println("nlength = " + nlength);
   double doublearray[] = new double[nlength];
   for(int i=1; i <= nlength; i++)
   {
   doublearray[i-1] = doubleBuffer.get(i);
   }
   // arithmetic mean
   double arithmean = 0.0;
   for(int i=0; i < nlength; i++)
   {
   arithmean += doublearray[i];
   }
   arithmean = arithmean/nlength;

   // harmonic mean
   double harmmean = 0.0;
   double temp = 0.0;
   for(int i=0; i < nlength; i++)
   {
   temp += 1.0/doublearray[i];
   }
   harmmean = nlength/temp;

   // geometric mean
   double geommean = 1.0;
   for(int i=0; i < nlength; i++)
   {
  geommean *= doublearray[i];
  }
  geommean = Math.exp(Math.log(geommean)/nlength);

  buffer.flip();
  buffer.clear();
  intBuffer.put(0,nlength);
  doubleBuffer.put(1,arithmean);
  doubleBuffer.put(2,harmmean);
  doubleBuffer.put(3,geommean);
  sc.write(buffer);
  }

  public static void main(String[] args)
  {
  new MeanServer().start();
  }
}
```

Chapter 12

Java 2 Collection Frame Work

12.1 Introduction

The Collections API is a set of classes and interfaces that provide an implementation-independent framework for working with collections of objects. This API consists of 13 new classes and 10 new interfaces that have been added to the `java.util` package. These classes and interfaces provide support for generic collections, sets, bags, maps, lists and linked lists. These classes and interfaces can be easily extended to provide support for custom object collections. Some classes and interfaces from Java version 1.0 have been replaced by improved interfaces and classes in version 1.2. For example, the interface `Iterator` replaces the interface `Enumeration` and the class `Map` replaces the `Dictionary` class. The `Vector` class introduced in chapter 3 can be replaced by class `ArrayList`.

Java 2 Collection Frame Work provides

Containers, Iterators, Algorithms.

The new collections interfaces introduced with JDK 1.2 are as follows:

`Collection`	`defines methods that implement the concept of a group of objects, referred to as elements`
`List`	`extends the Collection interface to implement an ordered collection of objects`
`Set`	`extends the Collection interface to implement a finite mathematical set. Sets differ from lists in that they do not allow duplicate elements`
`SortedSet`	`a Set whose elements are sorted in ascending order`

Comparator	provides the compare() method for comparing the elements of a collection
Iterator	provides methods for iterating through the elements of a collection. The Iterator interface replaces the Enumeration interface
ListIterator	extends the Iterator interface to support bidirectional iteration of lists
Map	replaces the Dictionary class as a means to associate keys with values
SortedMap	a Map whose elements are sorted in ascending order
Map.Entry	an inner interface of the Map interface that defines methods for working with a single key-value pair

The new collections classes introduced with JDK 1.2 are as follows:

AbstractCollection	The AbstractCollection class provides a basic implementation of the Collection interface. It is extended by other classes that tailor AbstractCollection to more specific implementations
AbstractList	The AbstractList class extends the AbstractCollection class to povide a basic implementation of the List interface
AbstractSequentialList	The AbstractSequentialList class extends the AbstractList class to provide a list that is tailored to sequential access, as opposed to random access
LinkedList	The LinkedList class extends the AbstractSequentialList class to provide an implementation of a doubly linked list. A doubly linked list is a list in which each element references both the previous and next elements in the list
ArrayList	The ArrayList class extends AbstractList to implement a resizeable array

AbstractSet	The AbstractSet class extends the AbstractCollection class to provide a basic implementation of the Set interface
HashSet	The HashSet class extends AbstractSet to implement a set of key-value pairs. It does not allow the use of the null element
TreeSet	The TreeSet class extends AbstractSet to implement the Set interface using a TreeMap
AbstractMap	The AbstractMap class provides a basic implementation of the Map interface
HashMap	The HashMap class extends AbstractMap to implement a hash table that supports the Map interface
WeakHashMap	The WeakHashMap class extends AbstractMap to implement a hash table that supports the Map interface and allows its keys to be garbage collected when no longer in ordinary use
TreeMap	The TreeMap class extends AbstractMap to implement a sorted binary tree that supports the Map interface
Arrays	The Arrays class provides static methods for searching and sorting arrays and converting them to lists
Collections	The Collections class provides static methods for searching, sorting and performing other operations on objects that implement the Collection interface.

12.2 Collection and Collections

The

```
public abstract interface Collection
```

is the root interface in the collection hierarchy. The

```
public abstract List extends Collection
```

is an ordered collection (also known as sequence). The user of this interface has precise control over where in the `List` each element is inserted. The user can access elements in the list by their integer index (position) and search for elements in the list. Unlike sets, lists can have duplicate elements. The

```
public class ArrayList
```

is a resizeable array implementation of the List interface. The constructor

```
ArrayList()
```

constructs an empty `ArrayList`. It implements all optional `List` operations, and permits all elements, including `null`. The method

```
boolean add(Object o)
```

in class `ArrayList` appends the specified element to the end of this `ArrayList`. The method

```
void add(int index,Object elem)
```

inserts the specified element at the specified position. To find out whether an element is the `ArrayList` we apply method

```
boolean contains(Object elem)
```

It returns `true` if this `ArrayList` contains the specified element. The method

```
Object get(int index)
```

returns the element at the specified position at this `ArrayList`. To remove all elements in `ArrayList` we use the method `void clear()`.

The class

```
public class Collections extends Objects
```

consists of static methods that operates on or return `Collections`. It contains polymorphic algorithms that operate on collections. The method

```
static void sort(List list)
```

sorts the specified `List` into ascending order, according to the natural ordering of its elements. The method

```
static int binarySearch(List list,Object key)
```

searches the specified `List` for the specified `Object` using the `binarySearch` algorithm. The method

```
static void copy(List dest,List src)
```

copies all the elements from one `List` into another. The method

```
static void fill(List list,Object o)
```

replaces all the elements of the specified `List` with the specified elements.

`Iterator` takes the place of `Enumeration` in the Java `Collections` framework. Iterators differ from Enumerations in two ways. Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics. Method names have been improved. The method

```
boolean hasNext()
```

in class `Iterator` returns `true` if the iteration has more elements. In other words it returns `true` if `next()` would return an element rather than throwing an exception. The method

```
Object next()
```

returns the next element in the iteration or `throws` an `NoSuchElementException` (iteration has no more elements). The method

```
void remove()
```

removes from the underlying `Collection` the last element returned by the `Iterator` (optional operation). This method can be called only once per call to `next()`.

12.3 Examples

The following two examples show how `Collection`, `ArrayList`, and `Iterator` is applied.

```
// MyColl1.java

import java.util.*;

public class MyColl1
{
   public static void main(String[] args)
   {
   Collection coll = new ArrayList();
   int i;
   for(i=65; i < 91; i++)
   {
   char c = (char) i;
   Character Ch = new Character(c);
   coll.add(Ch.toString());
   }

   Iterator it = coll.iterator();

   while(it.hasNext())
   {
   System.out.print(it.next()); // => ABC...YZ
   }
   System.out.println(" ");

   Collection doll = new ArrayList();
   Double D = new Double(3.14);
   Integer I = new Integer(45);
   Boolean B = new Boolean(true);

   doll.add((Double) D);    // type conversion necessary
   doll.add((Integer) I);
   doll.add((Boolean) B);

   Boolean B1 = (Boolean) ((List) doll).get(2);
   System.out.println("B1 = " + B1);

   } // end main
}
```

```
// MyColl2.java

import java.util.*;

public class MyColl2
{
   public static void main(String[] args)
   {
   Collection coll = new ArrayList();

   String s1 = new String("dulli");
   coll.add(s1);
   String s2 = new String("alli");
   coll.add(s2);
   String s3 = new String("bullo");
   coll.add(s3);
   String s4 = new String("xulli");
   coll.add(s4);

   int size = coll.size();
   System.out.println("size = " + size);

   Collections.sort((List) coll);

   Iterator itnew = coll.iterator();

   while(itnew.hasNext())
   {
   System.out.println(itnew.next()); // sorted list
   }

   int result = Collections.binarySearch((List) coll,"bullo");
   System.out.println("result = " + result); // result = 1
   }  // end main
}
```

12.4 Arrays Class

The

```
public class Arrays extends Objects
```

contains various methods for manipulating arrays such as sorting and searching. The methods work for basic data types and abstract data types. We apply the methods

```
sort(), binarySearch(), equals(), fill()
```

to arrays of basic data types and arrays of abstract data types. Before we call the method `binarySearch()` we have to sort the array using the `sort()` method. We also show how the method `clone()` is used.

```
// MyArray.java

import java.util.*;
import java.math.*;

public class MyArray
{
   public static void main(String[] s)
   {
   int i;

   int[] a = { 4, 5, -2, 3, 7, 9, 13, -1 };
   int[] b = { 3, 1, -8, 9, 11, 6, -6, 9 };

   Arrays.sort(a);
   for(i=0; i < a.length; i++)
   System.out.println("a[" + i + "] = " + a[i]);

   int r1 = Arrays.binarySearch(a,-1);
   System.out.println("r1 = " + r1);    // => 1

   boolean b1 = Arrays.equals(a,b);
   System.out.println("b1 = " + b1);   // => false

   double[] x = new double[5];
   Arrays.fill(x,3.14159);
   for(i=0; i < x.length; i++)
   System.out.println("x[" + i + "] = " + x[i]);

   boolean[] b2 = new boolean[6];
```

```
   Arrays.fill(b2,2,4,true);

   for(i=0; i < b2.length; i++)
   System.out.println("b2[" + i + "] = " + b2[i]);

   BigInteger[] big = new BigInteger[3];
   big[0] = new BigInteger("234567");
   big[1] = new BigInteger("6712345678");
   big[2] = new BigInteger("981");

   Arrays.sort(big);
   BigInteger f = new BigInteger("6712345678");
   int pos = Arrays.binarySearch(big,f);
   System.out.println("pos = " + pos);   // => 2

   for(i=0; i < big.length; i++)
   System.out.println("big[" + i + "] = " + big[i]);

   BigDecimal[] bigdec = new BigDecimal[4];
   bigdec[0] = new BigDecimal("4.567");
   bigdec[1] = new BigDecimal("-345.23456");
   bigdec[2] = new BigDecimal("9876.234567");
   bigdec[3] = new BigDecimal("1.23456789");

   Arrays.sort(bigdec);

   for(i=0; i < bigdec.length; i++)
   System.out.println("bigdec[" + i + "] = " + bigdec[i]);

   // use of clone()
   int[] vec = new int[3];
   vec[0] = 1; vec[1] = 4; vec[2] = 5;
   int[] vecClone = (int[]) vec.clone();
   for(i=0; i < vecClone.length; i++)
   System.out.println(vecClone[i]);
   }
}
```

The following program shows how an array of strings can be sorted.

```
// Sort.java

import java.util.*;

public class Sort
{
   public static void main(String[] args)
   {
   String[] strings =
   { "Willi", "Hans", "Sue", "Helen", "Nela" };

   Arrays.sort(strings);
   List list = Arrays.asList(strings);
   displayList(list);
   } // end main

   static void displayList(List list)
   {
   System.out.println("The size of the list is: " + list.size());

   ListIterator i = list.listIterator(0);
   while(i.hasNext())
   {
   Object o = i.next();
   if(o == null) System.out.println("null");
   else
   System.out.println(o.toString());
   } // end while
   } // end method display
}
```

The following program shows an application of the methods `shuffle`, `reverse`, `swap`, `fill`, `frequency` and `disjoint` in `Collections`.

```
// Sorting.java

import java.util.*;
import java.math.*;

public class Sorting
{
   public static void main(String[] args)
   {
```

```
   List<String> listS = Arrays.asList(args);
   Collections.sort(listS);
   System.out.println(listS);

   List<Integer> listI = new ArrayList<Integer>();
   listI.add(new Integer(11));
   listI.add(new Integer(14));
   listI.add(new Integer(3));
   Collections.sort(listI);
   System.out.println(listI);

   List<Object> listO = new ArrayList<Object>();
   listO.add(new Double(3.14));
   listO.add(new Character('x'));
   listO.add(new BigInteger("34567890"));
   System.out.println(listO);

   Collections.shuffle(listO);
   System.out.println(listO);

   Collections.reverse(listO);
   System.out.println(listO);

   Collections.swap(listO,2,0);
   System.out.println(listO);

   Collections.fill(listI,5);
   System.out.println(listI);

   int f = Collections.frequency(listI,5);
   System.out.println(f);

   List<Character> listC1 = new ArrayList<Character>();
   listC1.add(new Character('y'));
   listC1.add(new Character('x'));
   List<Character> listC2 = new ArrayList<Character>();
   listC2.add(new Character('a'));
   listC2.add(new Character('y'));
   boolean b = Collections.disjoint(listC1,listC2);
   System.out.println(b);

   int pos = Collections.binarySearch(listI,new Integer(5));
   System.out.println(pos);
   }
}
```

12.5 Class TreeSet

The class

```
public class TreeSet extends AbstractSet
implements SortedSet, Cloneable, Serializable
```

implements the `TreeSet` interface backed by `TreeMap`. The set will be in ascending order sorted according to the natural order of the elements.

The following program shows an application of the class `TreeSet`. The default constructor

```
TreeSet()
```

constructs a new empty `TreeSet` sorted according to the elements natural order. The method

```
boolean add(Object o)
```

adds the specified element to this `Set` if it is not already present.

The method

```
void clear()
```

removes all of the elements from the set.

The method

```
boolean contains(Object o)
```

returns `true` if this `TreeSet` contains the specified element.

The method

```
Object first()
```

returns the first (lowest) element currently in the `TreeSet`.

The method

```
Object last()
```

returns the last (highest) element currently in the `TreeSet`.

```
// MySet.java

import java.util.*;

public class MySet
{
    public static void main(String [] args)
    {
    String s1 = new String("xxyy");
    String s2 = new String("123");

    TreeSet tset = new TreeSet();
    tset.add(s1);
    tset.add(s2);

    System.out.println("The size of the set is: " + tset.size());

    Iterator it = tset.iterator();
    while(it.hasNext())
    {
    Object o = it.next();
    if(o == null) System.out.println("null");
    else
    System.out.println(o.toString());
    }

    String s3 = new String("a$$a");
    String s4 = new String("123");

    tset.add(s3);
    tset.add(s4);

    Iterator it1 = tset.iterator();
    while(it1.hasNext())
    {
    Object o = it1.next();
    if(o == null) System.out.println("null");
    else
    System.out.println(o.toString());
    }

    } // end main
}
```

In Java the interface `Set` is a `Collection` that cannot contain duplicates elements. The interface `Set` models the mathematical set abstraction. The `Set` interface extends `Collection` and contains no methods other than those inherited from `Collection`. It adds the restriction that duplicate elements are prohibited. The JDK contains two general-purpose `Set` implementations. The class `HashSet` stores its elements in a hash table. The class `TreeSet` stores its elements in a red-black tree. This guarantees the order of iteration.

The following program shows an application of the `TreeSet` class.

```
// SetOper.java

import java.util.*;

public class SetOper
{
   public static void main(String[] args)
   {
   String[] A = { "Steeb", "C++", "80.00" };
   String[] B = { "Solms", "Java", "80.00" };

   TreeSet S1 = new TreeSet();
   for(int i=0; i < A.length; i++)
   S1.add(A[i]);
   System.out.println("S1 = " + S1);

   TreeSet S2 = new TreeSet();
   for(int i=0;i < B.length;i++)
   S2.add(B[i]);
   System.out.println("S2 = " + S2);

   // union
   TreeSet S3 = new TreeSet(S1);
   boolean b1 = S3.addAll(S2);
   System.out.println("S3 = " + S3);
   System.out.println("S1 = " + S1);

   // intersection
   TreeSet S4 = new TreeSet(S1);
   boolean b2 = S4.retainAll(S2);
   System.out.println("S4 = " + S4);
   System.out.println("S2 = " + S2);

   // (asymmetric) set difference
```

```
   TreeSet S5 = new TreeSet(S1);
   boolean b3 = S5.removeAll(S2);
   System.out.println("S5 = " + S5);

   // test for subset
   TreeSet S6 = new TreeSet(S1);
   boolean b4 = S6.containsAll(S2);
   System.out.println("b4 = " + b4);

   // is element of set (contains)
   boolean b = S1.contains("80.00");
   System.out.println("b = " + b);
   b = S2.contains("Steeb");
   System.out.println("b = " + b);
   }
}
```

The output is

```
S1 = [80.00, C++, Steeb]
S2 = [80.00, Java, Solms]
S3 = [80.00, C++, Java, Solms, Steeb]
S1 = [80.00, C++, Steeb]
S4 = [80.00]
S2 = [80.00, Java, Solms]
S5 = [C++, Steeb]
b4 = false
b = true
b = true
```

12.6 Class TreeMap

The `TreeMap` class extends `AbstractMap` to implement a sorted binary tree that supports the `Map` interface. This impementation is not synchronized. If multiple threads access a `TreeMap` concurrently and at least one of the threads modifies the `TreeMap` structurally it must be synchronized externally.

The next program shows an application of the class `TreeMap`. The default constructor `TreeMap()` constructs a new, empty `TreeMap` sorted according to the keys in natural order. The constructor `TreeMap(Map m)` constructs a new `TreeMap` containing the same mappings as the given `Map`, sorted according to the key's natural order.

The method

```
Object put(Object key,Object value)
```

associates the specified value with the specified key in this `TreeMap`. The method

```
public Set entrySet()
```

in class `TreeMap` returns a `Set` view of the mapping contained in this map. The `Set`'s `Iterator` will return the mappings in ascending `Key` order. Each element in the returned set is a `Map.Entry`. The method

```
boolean containsKey(Object key)
```

returns `true` if this `TreeMap` contains a mapping for the specified key. The method

```
boolean containsValue(Object value)
```

returns `true` if this `Map` maps one or more keys to the specified value. The method

```
Object get(Object key)
```

returns the value to which this `TreeMap` maps the specified key. The method

```
Object remove(Object key)
```

removes the mapping for this key from this `TreeMap` if present.

```
// MyMap.java

import java.util.*;

public class MyMap
{
    public static void main(String[] args)
    {
    String[] sc = new String[3];
    sc[0] = new String("A");
    sc[1] = new String("B");
    sc[2] = new String("C");

    String[] sn = new String[3];
    sn[0] = new String("65");
    sn[1] = new String("66");
    sn[2] = new String("67");

    TreeMap map = new TreeMap();

    int i;
    for(i=0; i < sc.length; i++)
    {
    map.put(sc[i],sn[i]);
    }

    displayMap(map);
    }  // end main

    static void displayMap(TreeMap map)
    {
    System.out.println("The size of the map is: " + map.size());
    Collection c = map.entrySet();
    Iterator it = c.iterator();

    while(it.hasNext())
    {
    Object o = it.next();
    if(o == null)
    System.out.println("null");
    else
    System.out.println(o.toString());
    }
    } // end method displayMap
}
```

Chapter 13

The Swing Components

13.1 Introduction

The Swing components form a sophisticated library of GUI components including borders, buttons, checkboxes, combo boxes, icons, labels, lists, list boxes, menus, menubars, menu items, popup menus, radio buttons, progress bars, scrool panes and viewports, scrollbars, tabbed panes, tables, text areas, text components, text fields, trees and HTML viewers. All the components are Java Beans written in pure Java and all components can be navigated with the mouse as well with the keyboard. The Swing components extend the AWT (Abstract Windowing Toolkit) components. Swing components are used to create graphical user interfaces for Java programs. It includes not only the equivalent of AWT components, but also new ones such as table and treewidgets. Swing is a standard part of Java 2 and higher.

Swing offers a number of advantages over the AWT components. Just as AWT provides a Window class hierarchy, so does Swing. Swing's window classes are extensions of the AWT Window class hierarchy. The `JWindow` class extends the AWT `Window` class. The `JFrame` class extends the AWT `Frame` class and `JDialog` class extends the AWT `Dialog` class. The `JWindow`, `JFrame`, and `JDialog` classes differ from their AWT counterparts in that they use a separate content pane for adding and laying out GUI components. This content pane is a `Container` object that is accessed using the `getContentPane()`. The method

```
Container getContentPane()
```

is in class `JFrame` and returns the content pane object for this frame. The content pane is part of a `JRootPane` object that contains other panes for overlaying components and intercepting mouse and keyboard events. Thus the `JFrame` class is slightly incompatible with `java.awt.Frame`. The class `JFrame` contains a `JRootPane` as its only child. The `contentPane` should be the parent of any child of the `JFrame`. This is different from `java.awt.Frame`, e.g. to add a child to an AWT `Frame` we have

```
frame.add(child);
```

Using the class `JFrame` we have to add the child to the JFrames contentPane instead

```
frame.getContentPane().add(child);
```

The same is true for setting LayoutManagers, removing components, listing children, etc. All these methods should normally be sent to the `contentPane()` instead of the `JFrame` itself. Thus in Swing, top-level windows (such as `JFrame` or `JDialog`) do not have components added directly to the window. Instead we add them to the content pane like this

```
myWindow.getContentPane().add(component)
```

Swing menus, like Swing windows, are analogous to their AWT counterparts. The classes

```
JMenuBar, JMenu, JMenuItem, JCheckBoxMenuItem, JRadioButtonMenuItem
```

are used in the same manner as the AWT classes

```
MenuBar, Menu, MenuItem, CheckBoxMenuItem
```

but with one difference. The Swing menu classes are all subclasses of the `JComponent` class, and therefore, of the `Component` class. This means that Swing menus, unlike their AWT counterparts, are first-class components and can be used with any Container classes. The `JPopupMenu` class is analogous to the AWT `PopupMenu` class. The class `JPopupMenu` is an implementation of a Popup Menu – a small window which pops up and displays a series of choices.

The `JTextComponent`, `JTextField`, and `JTextArea` classes are the Swing analogs of the AWT `TextComponent`, `TextField`, and `TextArea` classes. In addition, Swing provides the `JTextPane` class for working with text documents that can be marked up with different text styles. The class `JTextPane extends JEditorPane` provides a text component that can be marked with attributes that are represented graphically. Components and images may be embedded in the flow of text.

The command in AWT

```
setLayout(new FlowLayout());
```

is replaced by

```
getContentPane().setLayout(new FlowLayout());
```

in swing.

The following three programs show an application of the classes `JButton`, `JTextField`, `JFrame`. We also show how the `UIManager` is used.

The method

```
String getText()
```

is in class `AbstractButton` and the class `JButton` inherts this method. The method

```
void setText(String)
```

is in class `JTextComponent` and the class `JTextField` inherts this method

If we want to change the look and feel of our application from the default, we can use the `UIManager` class which allows us to find out the current look and feel of the application, and to change it. The class `UIManager` keeps track of the current look and feel and its defaults. We can select from a variety of looks and feels for our user interface. The method

```
static void setLookAndFeel(LookAndFeel newLookAndFeel)
```

in class `UIManager` is passed a `LookAndFeel` object. The method

```
static void setLookAndFeel(String className)
```

in class `UIManager` is passed the name of the class as a `String` which represents the required look and feel. The most common way of using these methods is when we want to change from the cross-platform (metal) look and feel to the system look and feel.

13.2 Examples

In the next program we have rewritten the program `GUI.java` from chapter 5 now using Java 1.2. The class `WindowAdapter` is an abstract adapter class for receiving window events. The methods in this class are empty. This class exists as convenience for creating listener objects. We extend this class to create a `WindowEvent` Listener and override the methods for the events of interest. We create two buttons and add them to the container. We also add two checkboxes and three labels to the container. Finally a textfield is added.

```
// JGUI1.java

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class JGUI1 extends JFrame
{
  public JGUI1()
  {
  setSize(200,200);
  addWindowListener(new WindowAdapter()
  { public void windowClosing(WindowEvent event)
  { System.exit(0); }});

  setTitle("JGUI and its component");
  getContentPane().setLayout(new FlowLayout());

  JButton b1 = new JButton("Button 1");
  JButton b2 = new JButton("Button 2");
  getContentPane().add(b1);
  getContentPane().add(b2);

  JCheckBox ch1 = new JCheckBox("one",null,true);
  JCheckBox ch2 = new JCheckBox("two");
  JCheckBox ch3 = new JCheckBox("three");
  getContentPane().add(ch1);
  getContentPane().add(ch2);
  getContentPane().add(ch3);

  JLabel l1 = new JLabel("Hello Egoli");
  JLabel l2 = new JLabel("Good Night");
  getContentPane().add(l1);
  getContentPane().add(l2);
```

```
  JTextField t = new JTextField(20);
  getContentPane().add(t);
  }  // end constructor JGUI1

  public static void main(String []args)
  { new JGUI1().setVisible(true);  }
}
```

The program `MyJList.java` creates a simple list box example using the Swing API. This example generates a static list from which the user can make a selection.

```
// MyJList.java

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class MyJList extends JFrame
{
   private JPanel topPanel;
   private JList listbox;

   public MyJList()
   {
   setTitle("Simple ListBox Application");
   setSize(300,100 );
   addWindowListener(new WindowAdapter()
   { public void windowClosing(WindowEvent event)
   { System.exit(0); }});
   setBackground(Color.gray);
   topPanel = new JPanel();
   topPanel.setLayout(new BorderLayout());
   getContentPane().add(topPanel);
   // create some items to add to the list
   String listData[] = { "Item 1","Item 2","Item 3","Item 4" };

   listbox = new JList(listData);
   topPanel.add(listbox,BorderLayout.CENTER);
   }

   public static void main(String args[])
   {
   MyJList mainFrame = new MyJList();
   mainFrame.setVisible(true);
   }
}
```

13.3 Adding Actions to Containers

All Java events generated are subclasses of `java.awt.event`, and contain information such as the position where the event was caused and the source (the class where the event is generated). Different events classes can be used, for example a `MouseEvent` is generated when the mouse is used. If a programmer is only interested in the mouse when it is used to pressa button (and not in whether it is the left or right mouse button that has been pressed), they can simply use an `ActionEvent`. A piece of code that needs to know if an event happens should implement an `EventListener` and register it with components that may generate an event. If an event is generated the listener is called with the event as a parameter.

In the next two programs we include the interface `ActionListener` and the class `ActionEvent`. The method

```
Object getSource()
```

in class `ActionEvent` is inherited from class `EventObject`. The method

```
String getText()
```

returns the button's text. The class `JTextField` inherits the method

```
void setText(String t)
```

from class `JTextComponent`.

In the following program we define an `ActionListener` and register it on two buttons. When the mouse is used to press a button on the screen the buttons listen for mouse events and forward ActionEvents to the listener. We can run the program either as a Java application or as an Applet.

```
// Swing1.java

import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import javax.swing.*;
```

```
public class Swing1 extends Applet
{
   JButton b1 = new JButton("JButton 1");
   JButton b2 = new JButton("JButton 2");
   JTextField t = new JTextField(20);

   public void init()
   {
   ActionListener al = new ActionListener()
   {
   public void actionPerformed(ActionEvent e)
   {
   String name = ((JButton) e.getSource()).getText();
   t.setText(name + " Pressed");
   }
   };

   b1.addActionListener(al);
   add(b1);
   b2.addActionListener(al);
   add(b2);
   add(t);
   }  // end init()

   public static void main(String args [])
   {
   Swing1 applet = new Swing1();
   JFrame frame = new JFrame("TextAreaNew");
   frame.addWindowListener(new WindowAdapter()
   {
   public void windowClosing(WindowEvent e)
   { System.exit(0); }});

   frame.getContentPane().add(applet,BorderLayout.CENTER);
   frame.setSize(300,100);
   applet.init();
   applet.start();
   frame.setVisible(true);
   }  // end main
}
```

The `validate()` method is used to cause a container to lay out its subcomponents again. It should be invoked when this container's subcompoents are modified (added or removed from the container, or layout-related information changed) after the container has been displayed. An example is given below.

```
// Swing2.java

import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import javax.swing.*;

public class Swing2 extends Applet
{
   JButton b1 = new JButton("JButton 1");
   JButton b2 = new JButton("JButton 2");
   JTextField t = new JTextField(20);

   public void init()
   {
   ActionListener al = new ActionListener()
   {
   public void actionPerformed(ActionEvent e)
   {
   if(e.getSource() == b1)
   {
   JTextField t1 = new JTextField(5);
   add(t1);
   validate();
   }
   }
   };

   b1.addActionListener(al);
   add(b1);
   b2.addActionListener(al);
   add(b2);
   add(t);
   }  // end init()

   public static void main(String args [])
   {
   Swing2 applet = new Swing2();
   JFrame frame = new JFrame("TextAreaNew");
   frame.addWindowListener(new WindowAdapter()
   {
   public void windowClosing(WindowEvent e)
   { System.exit(0); }});

   frame.getContentPane().add(applet,BorderLayout.CENTER);
```

```
   frame.setSize(300,100);
   applet.init();
   applet.start();
   frame.setVisible(true);
   }  // end main
}
```

In the following program we open a window with a button which includes the text "Press me". After clicking the button it is replaced by a button with the text "1: I like this: Press me again". Clicking again overides the text in the button with the text 2: I like this: Press me again" etc up to 10. Every time the colour of the text changes in a random fashion.

```
// ButtonPress.java

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ButtonPress extends JFrame implements ActionListener
{
   private int red = 0, green = 0, blue = 0;
   private JButton button = new JButton("Press me!");
   private java.util.Random randomNumberGenerator =
         new java.util.Random();

   public ButtonPress(ButtonPressHandler buttonHandler)
   {
   setTitle("ButtonPress");

   addWindowListener(new WindowAdapter()
   { public void windowClosing(WindowEvent event)
   { System.exit(0); } } );

   getContentPane().setBackground(Color.white);
   getContentPane().setLayout(new FlowLayout());
   getContentPane().add(button);
   button.addActionListener(buttonHandler);
   button.addActionListener(this);
   setSize(400,150);
   setVisible(true);
   }  // end constructor ButtonPress

   public void actionPerformed(ActionEvent event)
   {
```

```
   red = randomNumberGenerator.nextInt(256);
   green = randomNumberGenerator.nextInt(256);
   blue = randomNumberGenerator.nextInt(256);
   button.setForeground(new Color(red,green,blue));
   }  // end method actionPerformed

   public static void main(String[] args) throws Exception
   {
   UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
   new ButtonPress(new ButtonPressHandler());
   }  // end main
}  // end class ButtonPress

class ButtonPressHandler implements ActionListener
{
   private int numPresses = 0;
   public void actionPerformed(ActionEvent event)
   {
   ++numPresses;
   ((JButton) event.getSource()).setText
      (numPresses + ":  I like this. Press me again.");
   if(numPresses == 10)
     ((JButton) event.getSource()).removeActionListener(this);
   }
}  // end class ButtonHandler
```

13.4 Button Game

The next program rewrites the program `ButtonGame` from chapter 5 using Swing components. We use the method `getContentPane()`. We replace the methods `setLabel()` and `getLabel()` from the class `Button` by the methods `setText()` and `getText()` from the class `java.swing.AbstractButton`.

```
// JButtonGame.java

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class JButtonGame extends JFrame implements ActionListener
{
      private int nRows, nCols, nButtons;
      private int blankCol, blankRow, clickedRow, clickedCol;
      private JButton[][] buttons;
      private JTextField textField;

      public JButtonGame()
      {
      setSize(200,200);
      addWindowListener(new WindowAdapter()
      { public void windowClosing(WindowEvent event)
      { System.exit(0);}});

      nRows = 4; nCols = 4; nButtons = nRows*nCols;
      JPanel panel = new JPanel();

      panel.setLayout(new GridLayout(nRows,nCols));
      buttons = new JButton[nRows][nCols];

      for(int nRow=0;nRow<nRows;nRow++)
      {
      for(int nCol=0;nCol<nCols;nCol++)
      {
      buttons[nRow][nCol] = new JButton ("");
      buttons[nRow][nCol].addActionListener(this);
      panel.add(buttons[nRow][nCol]);
      }
      }

      getContentPane().add("Center",panel);
      textField = new JTextField("",80);
```

```
textField.setEditable(false);
getContentPane().add("South",textField);

int labelsUsed[] = new int[nButtons];
for(int i=0; i<nButtons;i++)
{
boolean labelUsed;
int label;
do
{
label = random(nButtons) + 1;
labelUsed = false;
for(int j=0;j<i;j++)
labelUsed = ((labelUsed) || (label == labelsUsed[j]));
} while(labelUsed);

labelsUsed[i] = label;
int nRow = i/nCols;
int nCol = i - nRow*nCols;
buttons[nRow][nCol].setText((new Integer (label)).toString());
}
getButtonPosition((new Integer (nButtons)).toString());
blankRow = clickedRow;
blankCol = clickedCol;
JButton blank = buttons[clickedRow][clickedCol];
blank.setText("");
blank.setBackground(Color.green);
} // end constructor JButtonGame()

private int random(int k)
{
return (int) (k*Math.random() - 0.1);
} // end method random(int)

private void getButtonPosition(String label)
{
for(int nr=0;nr<nRows;nr++)
{
for(int nc=0;nc<nCols;nc++)
{
if(buttons[nr][nc].getText().equals(label))
{
clickedRow = nr;
clickedCol = nc;
textField.setText("[" + nr + ',' + nc + "]" + label);
```

```
        }
        }
        }
        } // end method getButtonPosition(String)

        public void actionPerformed(ActionEvent e)
        {
        getButtonPosition(e.getActionCommand());
        textField.setText("[" + blankRow + "," + blankCol
                        + "] = > [" + clickedRow + "," + clickedCol + "]");
        if(clickedRow == blankRow)
        {
        if(Math.abs(clickedCol - blankCol) == 1)
        moveBlank(blankRow,blankCol,clickedRow,clickedCol);
        }
        else
        if(clickedCol == blankCol)
        {
        if(Math.abs(clickedRow - blankRow) == 1)
        moveBlank(blankRow,blankCol,clickedRow,clickedCol);
        }
        } // end method actionPerformed

        public void moveBlank(int oldRow,int oldCol,int newRow,int newCol)
        {
        JButton oldBlank = buttons[oldRow][oldCol];
        JButton newBlank = buttons[newRow][newCol];

        String label = newBlank.getText();
        newBlank.setText("");
        newBlank.setBackground(Color.green);

        oldBlank.setText(label);
        oldBlank.setBackground(Color.lightGray);

        blankRow = newRow;
        blankCol = newCol;
        } // end method moveBlank

        public static void main(String[] args)
        {
        new JButtonGame().setVisible(true);
        }  // end method main
}
```

13.5 Editable Textfields

The following program shows how we implement editable textfields. The two textfields for the numerator and denominator are editable, whereas the textfield for the decimal approximation is non-editable. In the program we also implement *tool tips*, i.e. the extra hint popping up when the mouse hovers undecidedly over a component. All swing components inhert the ability of having tool tips associated to them from the mother of all swing components `JComponent`.

```
// TextFields.java

import java.awt.event.*;
import javax.swing.*;
import java.awt.*;
import java.text.DecimalFormat;

public class TextFields extends JFrame
             implements ActionListener, FocusListener
{
   private JTextField numField = new JTextField("1",20);
   private JTextField denField = new JTextField("1",20);
   private JTextField resultField = new JTextField("1",20);
   private DecimalFormat decimalFormat =
           new DecimalFormat("0.######E0");

   public TextFields()
   {
   addWindowListener(new WindowAdapter()
   { public void windowClosing(WindowEvent event)
   { System.exit(0); }
   });

   setTitle("Text Fields");

   getContentPane().setLayout(new GridLayout(3,2));
   getContentPane().add(new JLabel("Numerator: "));
   getContentPane().add(numField);
   getContentPane().add(new JLabel("Denominator: "));
   getContentPane().add(denField);
   getContentPane().add(new JLabel("Result of Division: "));
   getContentPane().add(resultField);

   numField.addActionListener(this);
   numField.addFocusListener(this);
   numField.setToolTipText("The Numerator Textfield");
```

```
   denField.addActionListener(this);
   denField.addFocusListener(this);
   denField.setToolTipText("The Denominator Textfield");
   resultField.setEditable(false);
   resultField.setToolTipText("The result of the divison.");

   pack();
   setVisible(true);
   } // end constructor TextFields()

   private void calculate()
   {
   double num = new Double(numField.getText()).doubleValue();
   double den = new Double(denField.getText()).doubleValue();
   double result = num/den;
   resultField.setText(decimalFormat.format(result));
   } // end calculate()

   public void actionPerformed(ActionEvent event) { calculate(); }
   public void focusLost(FocusEvent event) { calculate(); }
   public void focusGained(FocusEvent event) { }

   public static void main(String[] args)
   {
   new TextFields();
   }
}
```

13.6 Document and JTextPane Classes

In the next program (JDK Developer) we show how to apply the classes `Document` and `JTextPane`. We sets up a text window together with Set Position and Go To Position buttons. Then type some text into the text window. Next we move the text caret somewhere in that text. Then we click the button Set Position. This records this location in the text. Next we edit the text by adding and deleting characters before and after the position which was selected above. If we now click the Go To Position button the caret will move back to the location selected above. Thus the action of the Go To Position button takes into account the changed offset of the location based on edits we have done. The

```
public abstract interface Document
```

container for text supports editing and provides notification of changes. The class `JEditorPane` is sort of a fancy text area that can display text derived from different file formats. The built-in version supports HTML and RTF (Rich Text Format) only, but we can build editor kits to handle special purpose applications. JEditorPane is most often use to display HTML. We choose the type of document we want to display by calling the method `setContentType` and specify a custom editor kit via the method `setEditorKit`. The class

```
JTextPane extends JEditorPane
```

is a text component that can be marked up with attributes that are represented graphically. The class `JTextPane` inherits the method `getCaret()` from the class `JTextComponent`. The method `Caret getCaret()` fetches the caret that allows text-oriented navigation over the view. The

```
public abstract interface Caret
```

is a place within a document view that represents where things can be inserted into the document model. The method `Document getDocument()` in class `JTextPane` fetches the model associated with the editor. The

```
class Position
```

represents a location within a document. The method `int getDot()` in class `Caret` fetches the current position of the caret.

```
// PosDemo.java

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.text.*;
```

```
public class PosDemo
{
  static Position lastpos;

  public static void main(String args[])
  {
    JFrame frame = new JFrame("Position demo");
    frame.addWindowListener(new WindowAdapter() {
       public void windowClosing(WindowEvent e) {
         System.exit(0);
       }
       });  // end argument frame.addWindowListener

  final JTextPane pane = new JTextPane();
  pane.setPreferredSize(new Dimension(600,400));

  JButton setbutton = new JButton("Set Position");
  JButton gobutton = new JButton("Go To Position");

  setbutton.addActionListener(new ActionListener()
  {
    public void actionPerformed(ActionEvent e)
    {
    try
    {  // try block
       int dot = pane.getCaret().getDot();
       Document doc = pane.getDocument();
       lastpos = doc.createPosition(dot);
       pane.requestFocus();
    }  // end try block
    catch(BadLocationException exc)
    { System.err.println(exc); }
    }
    });  // end argument setbutton.addActionListener

  gobutton.addActionListener(new ActionListener()
  {
    public void actionPerformed(ActionEvent e)
    {
     if(lastpos == null)
     {
        Toolkit.getDefaultToolkit().beep();
     }
     else
       {
```

```
        int pos = lastpos.getOffset();
        pane.getCaret().setDot(pos);
        pane.requestFocus();
        }
    }
  });  // end argument gobutton.addActionListener

  JPanel buttonpanel = new JPanel();

  buttonpanel.add(setbutton);
  buttonpanel.add(gobutton);

  JPanel panel = new JPanel();

  panel.setLayout(new BorderLayout());
  panel.add("North", buttonpanel);
  panel.add("Center", pane);

  frame.getContentPane().add(panel);
  frame.pack();
  frame.setVisible(true);
  } // end main
}
```

13.7 Model-View-Controller

Swing's Model-UI architecture is a variation of the Model-View-Controller (MVC) pattern. MVC addresses the problem of how to build software for user interfaces. The MVC pattern, first introduced in Smalltalk-80, was designed to reduce the programming effort required to build systems that made use of multiple, synchronized presentations of the same data. The MVC consists of three distinct parts:

1) The *Model* holds the data that is being represented.

2) The *View* renders the data of the model. The view observes changes in the model and then renders the updated model data. Since there is only a single central model, all views will remain coherent.

3) The *Controller* listens to user events and controls the view on the data. The View and Controller know about the Model, but the model is almost unaware of their existence: it merely knows that if it changes, it has to send a message to any registered listener warning it that there have been changes.

The MVC pattern prompts the following software qualities, which make it such a novel idea:

1) Extensibility: It is common that the GUI of an application evolves much faster than its logic (the model). Using MVC, we may change the GUI without changing the model.

2) Loose coupling: The model publishes only an interface for notification. Each view implements this interface, but the model does not know, nor, care about the view beyond this interface.

3) Pluggabilty: We can add new views easily, without changing the model.

The MVC design pattern is typically used for constructing interfaces that may present multiple views of the same data. The design pattern is important, because it completely de-couples data display from data representation. This means that the user interface of an application can be completely changed - even dynamically, under direction of the user - without need for any change to the underlying data subsystem of the application. All Swing data-representation components are implemented with this design pattern in mind, so all Swing applications will inherently benefit from this immense design flexibility.

The easiest place to see this approch is the class `JTable`. We can define a `TableModel` object (the model) and tell a `JTable` (combined View-Controller) to observe that model. When the data changes, the `JTable` is notified, and it updates its display. Several Views could monitor a single table. The model notifies them of changes,

and lets the View figure out how to update the display. A common mistake is to write code to update the `TableModel` but not send any notifications.

```
// MyTable.java

import javax.swing.JTable;
import javax.swing.table.AbstractTableModel;
import javax.swing.JScrollPane;
import javax.swing.JFrame;
import javax.swing.SwingUtilities;
import javax.swing.JOptionPane;
import java.awt.*;
import java.awt.event.*;

public class MyTable extends JFrame
{
   private boolean DEBUG = true;

   public MyTable()
   {
   super("MyTable");
   MyTableModel myModel = new MyTableModel();
   JTable table = new JTable(myModel);
   table.setPreferredScrollableViewportSize(new Dimension(400,70));

   JScrollPane scrollPane = new JScrollPane(table);
   getContentPane().add(scrollPane,BorderLayout.CENTER);

   addWindowListener(new WindowAdapter()
   {
   public void windowClosing(WindowEvent e)
   {
   System.exit(0);
   }
   });

   } // end MyTable

   class MyTableModel extends AbstractTableModel
   {
   final String[] columnNames =
   { "First Name", "Last Name", "Sport", "# of Years", "Vegetarian" };

   final Object[][] data =
   {
```

```
{ "Mary", "Campione", "Snowboarding", new Integer(5),
   new Boolean(false) },
{ "Alison", "Huml", "Rowing", new Integer(3), new Boolean(true) },
{ "Kathy", "Walrath", "Tennis", new Integer(2), new Boolean(false) },
{ "Mark", "Andrews", "Boxing", new Integer(10), new Boolean(false) },
{ "Angela", "Lih", "Running", new Integer(5), new Boolean(true) }
};

public int getColumnCount()  { return columnNames.length; }

public int getRowCount()  { return data.length; }

public String getColumnName(int col)  { return columnNames[col]; }

public Object getValueAt(int row,int col)  { return data[row][col]; }

public Class getColumnClass(int c)
{ return getValueAt(0,c).getClass(); }

public boolean isCellEditable(int row,int col)
{
if(col < 2) { return false; }
else  { return true; }
}

public void setValueAt(Object value,int row,int col)
{
if(DEBUG)
{
System.out.println("Setting value at " + row + "," + col
                                        + " to " + value
                                        + " (an instance of "
                                        + value.getClass() + ")");
}

if(data[0][col] instanceof Integer)
{
try
{
data[row][col] = new Integer((String) value);
fireTableCellUpdated(row,col);
}
catch(NumberFormatException e)
{
JOptionPane.showMessageDialog(MyTable.this,
```

```
          "The \"" + getColumnName(col)
          + "\" column accepts only integer values.");
    }
    }
    else
    {
    data[row][col] = value;
    fireTableCellUpdated(row,col);
    }

    if(DEBUG)
    {
    System.out.println("New value of data:");
    printDebugData();
    }
    }

    private void printDebugData()
    {
    int numRows = getRowCount();
    int numCols = getColumnCount();

    for(int i=0;i<numRows;i++)
    {
    System.out.print("  row " + i + ".");
    for(int j=0;j<numCols;j++)
    {
    System.out.print("  " + data[i][j]);
    }
    System.out.println();
    }
    System.out.println("----------------------------------");
    }
    }

    public static void main(String[] args)
    {
    MyTable frame = new MyTable();
    frame.pack();
    frame.setVisible(true);
    }
}
```

13.8 JTree Class

Trees allow us to represent hierarchies of data in a convenient, accessible form. Swing's `JTree` component enables developers to easily capture (represent in a data structure) and display hierarchies. The class `JTree` implements a tree structure that can be used to display hierarchical data structures. The `TreeNode` interface defines methods that are to be implemented by the node of a `JTree` object. The

```
public class DefaultMutableTreeNode extends Object
implements Cloneable, MutableTreeNode, Serializable
```

provides a default implementation of the `TreeNode` interface. Trees are created by creating objects of the `TreeNode` interface and then adding them together using the `add()` method. When all of the `TreeNode` objects have been added together, the resulting `TreeNode` object is passed to the `JTree` constructor. A

```
DefaultMutableTreeNode
```

is a general purpose node in a tree data structure. A tree node may have at most one parent and 0 or more children. `DefaultMutableTreeNode` provides operations for examining and modifying node's parent and children and also operations for examining the tree that the node is part of. The constructor

```
DefaultMutableTreeNode(Object userObject)
```

creates a tree node with no parent, no children, but which allows children and initializes it with the specified user object. The method

```
void add(MutableTreeNode newChild)
```

removes `newChild` from its parent and makes it a child of this node by adding it to the end of this node's child array.

A single tree selection is represented by a `TreePath` object, which defines a path from a root node to the selected node. The method

```
Object getPathComponent(int element)
```

returns the path component at the specified index. The method

```
int getPathCount()
```

returns the number of elements in the path.

The class `JScrollPane` is a specialized container that manages a viewport, optional vertical and horizontal scrollbars, and optional row and column heading viewports.

The next program shows an application of the `JTree` class.

```
// JTreeClass.java

import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.tree.*;

public class JTreeClass extends JFrame
{
   public static int WIDTH = 400;
   public static int HEIGHT = 400;
   public static String TITLE = new String("JTree Application");

   Container frameContainer;

   JTextField textField = new JTextField();
   JScrollPane scrollPane = new JScrollPane();
   JTree tree;
   Renderer renderer = new Renderer();

   DefaultMutableTreeNode root = new DefaultMutableTreeNode("Willi");

   DefaultMutableTreeNode son1_willi =
   new DefaultMutableTreeNode("Lars");
   DefaultMutableTreeNode son2_willi =
   new DefaultMutableTreeNode("Gunnar");
   DefaultMutableTreeNode son3_willi =
   new DefaultMutableTreeNode("Jan");
   DefaultMutableTreeNode son4_willi =
   new DefaultMutableTreeNode("Olli");
   DefaultMutableTreeNode son1_Lars =
   new DefaultMutableTreeNode("Friederich");
   DefaultMutableTreeNode son1_Gunnar =
   new DefaultMutableTreeNode("Eugen");
   DefaultMutableTreeNode son2_Gunnar =
   new DefaultMutableTreeNode("Karl");
   DefaultMutableTreeNode son1_Jan =
   new DefaultMutableTreeNode("Albert");
   DefaultMutableTreeNode son1_Olli =
   new DefaultMutableTreeNode("Ollix");
   DefaultMutableTreeNode son1_Eugen =
```

```
new DefaultMutableTreeNode("Otto");

JMenuBar menuBar = new JMenuBar();
JMenu fileMenu = new JMenu("File");
JMenuItem fileExit = new JMenuItem("Exit");

public JTreeClass()
{
super(TITLE);
buildGUI();
setupEventHandlers();
setSize(WIDTH,HEIGHT);
show();
}

void buildGUI()
{
setupMenuBar();
setupTree();
layoutComponents();
}

void setupMenuBar()
{
fileMenu.add(fileExit);
menuBar.add(fileMenu);
setJMenuBar(menuBar);
}

void setupTree()
{
root.add(son1_willi);
root.add(son2_willi);
root.add(son3_willi);
root.add(son4_willi);
son1_willi.add(son1_Lars);
son2_willi.add(son1_Gunnar);
son2_willi.add(son2_Gunnar);
son3_willi.add(son1_Jan);
son4_willi.add(son1_Olli);
son1_Gunnar.add(son1_Eugen);

tree = new JTree(root);
}
```

```
public void layoutComponents()
{
frameContainer = getContentPane();
frameContainer.setLayout(new BorderLayout());
tree.setCellRenderer(renderer);
tree.addTreeSelectionListener(new TreeHandler());
scrollPane.getViewport().add(tree);
frameContainer.add("Center",scrollPane);
frameContainer.add("South",textField);
}

void setupEventHandlers()
{
addWindowListener(new WindowHandler());
fileExit.addActionListener(new MenuItemHandler());
}

public static void main(String[] args)
{
JTreeClass app = new JTreeClass();
}

public class WindowHandler extends WindowAdapter
{
public void windowClosing(WindowEvent e) { System.exit(0); }
} // end class WindowHandler

public class MenuItemHandler implements ActionListener
{
public void actionPerformed(ActionEvent e)
{
String cmd = e.getActionCommand();
if(cmd.equals("Exit")) System.exit(0);
}
}  // end class MenuItemHandler

public class TreeHandler implements TreeSelectionListener
{
public void valueChanged(TreeSelectionEvent e)
{
TreePath path = e.getPath();
String text =
path.getPathComponent(path.getPathCount()-1).toString();

textField.setText(text);
```

```
   }
   }

   class Renderer extends JLabel implements TreeCellRenderer
   {
   public Component
   getTreeCellRendererComponent(JTree tree,Object value,
   boolean selected,boolean expanded,boolean leaf,int row,
   boolean hasFocus)
   {
   setText(value.toString()+"                    ");
   return this;
   }
   }
}
```

13.9 Class JEditorPane

`JEditorPane` is a sort of text area that can display text derived from different file formats. The built-in version supports HTML and RTF (Rich Text Format) only, but we can build editor kits to handle special purpose applications. If we have plane text we could also use `JTextField` instead of `JEditorPane`. The following program shows an application of the class `JEditorPane`. A simple web browser is implemented.

```
// Browser.java

import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.io.*;

public class Browser extends JFrame
       implements HyperlinkListener, ActionListener
{
   private JIconButton homeButton;
   private JTextField urlField;
   private JEditorPane htmlPane;
   private String initialURL;

   public Browser(String initialURL)
   {
   super("Swing Browser");
   this.initialURL = initialURL;
   addWindowListener(new WindowAdapter()
   {
   public void windowClosing(WindowEvent event)
   { System.exit(0); }} );

   JPanel topPanel = new JPanel();
   topPanel.setBackground(Color.yellow);
   homeButton = new JIconButton("OUTPUT.JPG");
   homeButton.addActionListener(this);
   JLabel urlLabel = new JLabel("URL:");
   urlField = new JTextField(30);
   urlField.setText(initialURL);
   urlField.addActionListener(this);
   topPanel.add(homeButton);
   topPanel.add(urlLabel);
```

```
topPanel.add(urlField);
getContentPane().add(topPanel,BorderLayout.NORTH);

try
{
htmlPane = new JEditorPane(initialURL);
htmlPane.setEditable(false);
htmlPane.addHyperlinkListener(this);
JScrollPane scrollPane = new JScrollPane(htmlPane);
getContentPane().add(scrollPane,BorderLayout.CENTER);
}
catch(IOException ioe)
{
warnUser("Cannot build HTML pane for " + initialURL
         + ": " + ioe);
}

Dimension screenSize = getToolkit().getScreenSize();
int width = screenSize.width*8/10;
int height = screenSize.height*8/10;
setBounds(width/8,height/8,width,height);
setVisible(true);
} // end  constructor Browser(String initialURL)

public void actionPerformed(ActionEvent event)
{
String url;
if(event.getSource() == urlField)
url = urlField.getText();
else
url = initialURL;
try
{
htmlPane.setPage(new URL(url));
urlField.setText(url);
}
catch(IOException ioe)
{
warnUser("Cannot follow link to " + url + ": " + ioe);
}
} // end actionPerformed

public void hyperlinkUpdate(HyperlinkEvent event)
{
if(event.getEventType() == HyperlinkEvent.EventType.ACTIVATED)
```

```
   {
   try
   {
   htmlPane.setPage(event.getURL());
   urlField.setText(event.getURL().toExternalForm());
   }
   catch(IOException ioe)
   {
   warnUser("Cannot follow link to "
            + event.getURL().toExternalForm() + ": " + ioe);
   }
   }
   } //end hyperlinkUpdate

   private void warnUser(String message)
   {
   JOptionPane.showMessageDialog(this,message,"Error",
                                 JOptionPane.ERROR_MESSAGE);
   }

   public class JIconButton extends JButton
   {
   public JIconButton(String file)
   {
   super(new ImageIcon(file));
   setContentAreaFilled(false);
   setBorderPainted(false);
   setFocusPainted(false);
   }
   }

   public static void main(String[] args)
   {
   if(args.length == 0)
     new Browser("http://issc.rau.ac.za");
   else new Browser(args[0]);
   } // end main
}
```

13.10 Printing in Java

Rendering to a printer is like rendering to the screen. The printing system controls when pages are rendered, just like the drawing system controls when a component is painted on the screen. Our application provides the printig system with information about the document to a printed, and the printing system determines when each page needs to be imaged. When pages need to be imaged, the printing system calls our application's print method with an appropiate Graphics context. To use Java 2D API features when we print, we cast the `Graphics` object to a `Graphics2D`, just like we do when we are rendering to the screen.

The class `PrinterJob` is the principal class that controls printing. An application calls methods in this class to set up a job, optionally to invoke a print dialog with the user, and then print the pages of the job. The following methods are in this class.

The method

```
static PrinterJob getPrinterJob()
```

creates and returns a `PrinterJob`. The method

```
abstract void setPrintable(Printable painter)
```

is used when the pages in the document to be printed by this `PrinterJob` are drawn by the `Printable` object painter. The method

```
abstract boolean printDialog()
```

presents the user a dialog for changing properties of the print job interactivly. The method

```
abstract void print()
```

prints a set of pages.

```
// PrintTest.java

import javax.swing.*;
import java.awt.*;
import java.awt.geom.*;
import java.awt.event.*;
import java.awt.print.*;
import java.awt.print.PrinterJob;

public class PrintTest extends JFrame implements ActionListener
{
   public PrintTest()
```

```
   {
   addWindowListener(new WindowAdapter()
   {
   public void windowClosing(WindowEvent event)
   { System.exit(0); }
   });

   setTitle("Print Test");

   JPanel buttonPanel = new JPanel();
   buttonPanel.add(printButton);
   printButton.addActionListener(this);

   getContentPane().add(buttonPanel,BorderLayout.NORTH);
   getContentPane().add(drawPanel);

   setSize(400,400);
   setVisible(true);
   }

   public void actionPerformed(ActionEvent event)
   {
   if(event.getSource() == printButton)
   drawPanel.print();
   }

   public static void main(String[] args)
   {
   new PrintTest();
   }

   private JButton printButton = new JButton("Print");
   private DrawPanel drawPanel = new DrawPanel();
}  // end class PrintTest

class DrawPanel extends JPanel implements Printable
{
   public void paint(Graphics gc)
   {
   draw((Graphics2D) gc);
   }

   private void draw(Graphics2D gc)
   {
   Dimension panelSize = getSize();
```

```
   int width = (int) panelSize.getWidth();
   int height = (int) panelSize.getHeight();

   gc.setPaint(new GradientPaint(width/4.0F,height/2.0F,Color.black,
               3.0F*width/4.0F,height/2.0F,Color.white));

   Ellipse2D ellipse
    = new Ellipse2D.Double(width/4.0,height/4.0,width/2.0,height/2.0);
   gc.fill(ellipse);
   }

   public int print(Graphics g,PageFormat pf,int pi)
              throws PrinterException
   {
   if(pi >= 1)
     return Printable.NO_SUCH_PAGE;
   draw((Graphics2D) g);
   return Printable.PAGE_EXISTS;
   }

   public void print()
   {
   PrinterJob printJob = PrinterJob.getPrinterJob();
   printJob.setPrintable(this);
   if(printJob.printDialog())
   {
   try { printJob.print(); }
   catch (Exception e) { e.printStackTrace(); }
   }
   }
}  // end class DrawPanel
```

Chapter 14

Java Beans

14.1 Introduction

Java Beans is a component technology on top of the Java programming language that tries to simplify programming by reducing it as far as possible to simple graphical operations. It is used mainly to implement graphical user interfaces, but can be applied to the non-graphical parts of a program as well. A Java Bean is a reusable software component that works with Java. More specifically: a Java Bean is a reuseable software component that can be visually manipulated in builder tools. To work with Java Beans one needs an appropriate tool like a Java Integrated Development Environment and a set of components, the beans.

Individual Java Beans will vary in functionality, but share certain common defining features.

1) Support for introspection allowing a builder tool to analyze how a bean works.

2) Support for customization allowing a user to alter the appearance and behaviour of a bean.

3) Support for events allowing beans to fire events, and informing builder tools about both the events they can fire and the events they can handle.

4) Support for properties allowing beans to be manipulated programmatically, as well as to support the customization.

5) Support for persistence allowing beans that beans that have been customized in an application builder to have their state saved and restored. Typically persistence is used with an application builder's save and load menu commands to restore any work that has gone into constructing an application.

The most common use of beans is for graphical user interface components, such as components of the `java.awt` and `javax.swing` packages. Although all beans can be manipulated visually, this does not mean every bean has it own visual representation. The simplest beans are typically basic graphical interface components, such as a `java.awt.Button` object.

The Java Beans component model consists of a `java.beans` and `java.beans.beancontext` packages and a number of important naming and API conventions to which conforming beans and bean-manipulation tools must adhere. Since Java Beans is a framework for generic components, the Java Beans conventions are, in many ways, more important than the actual API.

The Java Bean APIs contain a `java.beans.Introspector` class which can be used to find out about the properties and methods of a class. It can find out about properties of a bean by either an implicit or explicit method. The implicit method uses design patterns. The Introspector creates a list of all of the public methods in the bean and searches that list for signatures that matches a particular pattern. The explicit introspection uses an array of package names to form a search path for locating explicit information about a bean.

Sometimes when a bean property changes, another object may want to be notified of the change and react to the change. Whenever a bound property changes, notification of the change is sent to interested listeners. A bean containing a bound property must maintain a list of property change listeners, and alert those listeners when the bound property changes. The convenience class `PropertyChangeSupport` implements methods that add and remove `PropertyChangeListener` objects from a list, and fires `PropertyChangeEvent` objects at those listeners when the bound property changes. Our beans can inherit from this class, or use it as an inner class. An object that wants to listen for property changes must be able to add and remove itself from the listener list on the bean containing the bound property, and respond to the event notification method that signals a property change. By implementing the `PropertyChangeListener` interface the listener can be added to the list maintained by the bound property bean, and since it implements the

```
void PropertyChangeListener.propertyChange(PropertyChangeEvent evt)
```

method, the listener can respond to property change notification. The

```
PropertyChangeEvent
```

class encapsulates property change information, and is sent from the property change event source to each object in the property change listener list via the `propertyChange` method.

14.2 Example

The following programs give an example for a bean. The class `DateAlerter` implements a date alerter and the class `DateWatcher` implements a date watcher. The class `DateAlerter` uses the `Date` class and we implement the public methods

```
synchronized void addPropertyChangeListener(PropertyChangeListener lis)
```

and

```
synchronized void removePropertyChangeListener(PropertyChangeListener lis)
```

We also use object serialization. Object serialization supports the encoding of objects and the objects reachable from them into a stream of bytes. It supports the complementary reconstruction of the object from the stream. Both `String` and `Date` implement serializable.

```
// TimeBeans.java

public class TimeBeans
{
  public static void main(String[] args)
  {
  // instaniate beans to watch and warn
  DateAlerter alert = new DateAlerter();
  DateWatcher watch = new DateWatcher(alert);
  long time = alert.getDateTime();
  // Set time forward by 10000 units
  alert.setDateTime(time + 10000);
  alert.showDateTime();
  System.exit(0);
  }
}

// DateAlerter.java

import java.beans.*;
import java.text.*;
import java.util.*;

public class DateAlerter extends Object
                         implements java.io.Serializable
{
  private static final String PROP_PROPERTY = "dateTime";
  private long dateTime;
  private Date aDate;
```

```
private PropertyChangeSupport propertySupport;

// Creates new DateAlerter
public DateAlerter()
{
propertySupport = new PropertyChangeSupport(this);
aDate = new Date();
dateTime = aDate.getTime();
System.out.println("Alerter Waiting");
}

public void showDateTime()
{
aDate = new Date();
String myDate = DateFormat.getDateInstance().format(aDate);
String myTime = DateFormat.getTimeInstance().format(aDate);
System.out.println("The Time is " + myDate + " " +  myTime);
}

public long getDateTime()
{
dateTime = aDate.getTime();
return dateTime;
}

public void setDateTime(long value)
{
long oldValue = dateTime;
dateTime = value;
aDate.setTime(dateTime);
if(propertySupport.hasListeners(PROP_PROPERTY))
{
Long WdateTime = new Long(dateTime);
Long WoldValue = new Long(oldValue);
propertySupport.firePropertyChange(PROP_PROPERTY,WoldValue,
                                   WdateTime);
}
}

public void addPropertyChangeListener(PropertyChangeListener lis)
{
propertySupport.addPropertyChangeListener(lis);
}

public void removePropertyChangeListener(PropertyChangeListener lis)
```

```
  {
  propertySupport.removePropertyChangeListener(lis);
  }
}

// DateWatcher.java

import java.beans.*;
import javax.swing.*;

public class DateWatcher extends Object
    implements java.io.Serializable, PropertyChangeListener
{
  DateAlerter myAlert;
  // Creates new Bean
  public DateWatcher(DateAlerter anAlert)
  {
  myAlert = anAlert;
  myAlert.addPropertyChangeListener(this);
  System.out.println("Watcher on Patrol");
  }

  public void propertyChange(PropertyChangeEvent evt)
  {
  JOptionPane.showMessageDialog(null,"System Date Modified",
          "ALERT",JOptionPane.PLAIN_MESSAGE);
  System.out.println("end");
  }
}
```

14.3 JavaBeans

JavaBeans are Java classes that follow conventions which allow for easy reuse of components. This allows, for example, the creation of new GUI components which can be used and manipulated from the NetBeans IDE GUI builder.

JavaBeans are not necessarily graphical in nature. The NetBeans IDE provides straightforward access to the capabilities of JavaBeans.

14.3.1 Creating a JavaBean: NetBeans

Add a new Java class to a NetBeans project. Properties can be added by right-clicking in the editor window, choosing "**Insert Code**" and "**Add Property**".

A bean can be used from the palette for a JFrame form. Right clicking on a JFrame form class node in a project and selecting "**Tools**" followed by "**Add to Palette**" adds a bean to the palette of beans available to the form.

14.3.2 Adding a JavaBean JAR: NetBeans

Beans stored in a JAR file can be used from the palette. Select the "**Tools**" menu, "**Palette**" and "**Swing/AWT Components**".

Select the "**Add from JAR...**" button. Select the JAR file and the beans to be used in the palette.

14.3.3 Simple properties

Any methods of a class of the form

SomeType `getSomething(void)`
`void setSomething(`*SomeType*`)`

describes a simple property named "`something`". If `setSomething` is missing, then "`something`" is a read-only property.

14.3.4 Simple properties

```
//Alarm1.java

import java.util.Timer;
import java.util.TimerTask;

public class Alarm1 extends TimerTask
{
 private int alarm; //milliseconds
 private int rings;
 private Timer alarmTimer;

 public Alarm1()
 {
  alarm = -1;
  rings = 0;
  alarmTimer = new Timer("Alarm", true); // is a daemon
 }

 public void run()
 {
  rings++;
```

```
  System.out.println("Alarm (" + rings + ") ...");
 }

 public int getAlarm() { return alarm; }

 public void setAlarm(int newalarm)
 {
  alarm = newalarm;
  if(alarm < 0) alarmTimer.cancel();
  else
  {
   alarmTimer = new Timer("Alarm", true); // is a daemon
   if(alarm >= 0) alarmTimer.scheduleAtFixedRate(this, alarm, alarm);
   else alarmTimer.cancel();
  }
 }

 public int getRings() { return rings; }

 public static void main(String[] args)
 {
  Alarm1 a = new Alarm1();
  a.setAlarm(1000); try { Thread.sleep(5000); }
  catch(java.lang.InterruptedException e) { }
 }
}
```

14.3.5 Beans should be serializable

In general, JavaBeans should implement `Serializable` so that the bean state can be saved and restored. This allows for the properties of a bean to be saved with a GUI design.

In many cases this is as simple as

```
public Class class ... implements Serializable
```

14.3.6 Creating a JAR file for JavaBeans

`jar cvmf` *manifest jarfile* files ...

```
jar cvmf Alarm1.manifest Alarm1.jar Alarm1.class
```

```
added manifest
adding: Alarm1.class(in = 1488) (out= 857)(deflated 42%)
```

```
Alarm1.manifest:

Manifest-Version: 1.0

Name: Alarm1.class
Java-Bean: true
```

14.3.7 Introspection

```
//ShowBean.java

import java.beans.BeanDescriptor;
import java.beans.BeanInfo;
import java.beans.IndexedPropertyDescriptor;
import java.beans.Introspector;
import java.beans.PropertyDescriptor;
import java.lang.ClassLoader;

public class ShowBean
{
 public static void main(String[] args)
 {
  if(args.length != 1)
   System.out.println("One argument: the name of the JavaBean class.");
  else try
  {
   System.out.println("Info for JavaBean: " + args[0]);
   ClassLoader cl = ClassLoader.getSystemClassLoader();
   BeanInfo bi = Introspector.getBeanInfo(cl.loadClass(args[0]));
   BeanDescriptor bd = bi.getBeanDescriptor();
   System.out.println("Name: " + bd.getName());
   System.out.println("Description: " + bd.getShortDescription());
   PropertyDescriptor[] pd = bi.getPropertyDescriptors();

   int i;
   for(i=0;i<pd.length;i++)
   {
    System.out.println("Property: " + pd[i].getName()
                       + ((pd[i].isBound())?" (bound)":"")
                       + ((pd[i].isConstrained())?" (constrained)":"")
                       + ((pd[i] instanceof IndexedPropertyDescriptor)?
                          " (indexed)":""));
    System.out.println("Property: " + pd[i].getName() + ", read: "
                                    + pd[i].getReadMethod());
    System.out.println("Property: " + pd[i].getName() + ", write: "
```

```
                                    + pd[i].getWriteMethod());
   }
  }
  catch(java.lang.ClassNotFoundException e)
  { System.out.println("Class not found: " + args[0]); }
  catch(java.beans.IntrospectionException e)
  { System.out.println("Error during introspection: " + args[0]); }
 }
}

java ShowBean Alarm1

Info for JavaBean: Alarm1
Name: Alarm1
Description: Alarm1
Property: alarm
Property: alarm, read: public int Alarm1.getAlarm()
Property: alarm, write: public void Alarm1.setAlarm(int)
Property: class
Property: class, read: public final native java.lang.Class
 java.lang.Object.getClass()
Property: class, write: null
Property: rings
Property: rings, read: public int Alarm1.getRings()
Property: rings, write: null
```

14.3.8 Bound properties

A bound property is a simple property that notifies listeners of property changes. This is achieved with an implementation (by inheritence or by aggregation) of `java.beans.PropertyChangeSupport`.

Changes are announced using the

`firePropertyChange(`"*name*"`,` *old value*`,` *new value*`)`

member of `PropertyChangeSupport`. The two members

```
void addPropertyChangeListener(PropertyChangeListener)
void removePropertyChangeListener(PropertyChangeListener)
```

must also be implemented. A `PropertyChangeListener` implements the `propertyChange(Property`
method.

```
//Alarm2.java
```

```
import java.beans.PropertyChangeEvent;
import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeSupport;
import java.util.Timer;
import java.util.TimerTask;

public class Alarm2 extends TimerTask implements PropertyChangeListener
{
 private int alarm; //milliseconds
 private int rings;
 private Timer alarmTimer;
 private PropertyChangeSupport pChange; // can also inherit

 public Alarm2()
 {
  alarm = -1;
  rings = 0;
  alarmTimer = new Timer("Alarm",true); // is a daemon
  pChange = new PropertyChangeSupport(this);
  pChange.addPropertyChangeListener(this);
 }

 // TimerTask
 public void run()
 {
  int oldrings = rings;
  rings++;
  pChange.firePropertyChange("rings",oldrings,rings);
 }

 // PropertyChangeListener
 public void propertyChange(PropertyChangeEvent e)
 {
  System.out.println(e.getPropertyName() + " changed from "
                     + e.getOldValue() + " to " + e.getNewValue());
 }

 // PropertyChangeSupport
 public void addPropertyChangeListener(PropertyChangeListener p)
 { pChange.addPropertyChangeListener(p); }

 public void removePropertyChangeListener(PropertyChangeListener p)
 { pChange.removePropertyChangeListener(p); }
```

```
 public int getAlarm() { return alarm; }

 public void setAlarm(int newalarm)
 {
  alarm = newalarm;
  if(alarm < 0) alarmTimer.cancel();
  else
  {
   alarmTimer = new Timer("Alarm", true); // is a daemon
   if(alarm >= 0) alarmTimer.scheduleAtFixedRate(this,alarm,alarm);
   else alarmTimer.cancel();
  }
 }

 public int getRings() { return rings; }

 public static void main(String[] args)
 {
  Alarm2 a = new Alarm2();
  a.setAlarm(1000); try { Thread.sleep(5000); }
  catch(java.lang.InterruptedException e) { }
 }
}
```

```
java ShowBean Alarm2
```

```
Info for JavaBean: Alarm2
Name: Alarm2
Description: Alarm2
Property: alarm (bound)
Property: alarm, read: public int Alarm2.getAlarm()
Property: alarm, write: public void Alarm2.setAlarm(int)
Property: class
Property: class, read: public final native java.lang.Class
 java.lang.Object.getClass()
Property: class, write: null
Property: rings (bound)
Property: rings, read: public int Alarm2.getRings()
Property: rings, write: null
```

14.3.9 Constrained properties

A constrained property is very similar to a bound property. However, listeners may throw a `PropertyVetoException` to indicate that a property change should not be allowed. This is achieved with an implementation of

`java.beans.VetoableChangeSupport`.

Changes are announced using the
`fireVetoableChange(`"*name*", *old value*, *new value*`)`
`throws PropertyVetoException`
member of `VetoableChangeSupport`. The two members
`void addVetoableChangeListener(PropertyChangeListener)`
`void removeVetoableChangeListener(PropertyChangeListener)`
must also be implemented. A `VetoableChangeListener` implements the
`vetoableChange(PropertyChangeEvent)`
`throws PropertyVetoException`
method.

```
//Alarm3.java

import java.beans.PropertyChangeEvent;
import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeSupport;
import java.beans.PropertyVetoException;
import java.beans.VetoableChangeListener;
import java.beans.VetoableChangeSupport;
import java.util.Timer;
import java.util.TimerTask;

public class Alarm3 extends TimerTask
 implements PropertyChangeListener, VetoableChangeListener
{
 private int alarm; //milliseconds
 private int rings;
 private Timer alarmTimer;
 private PropertyChangeSupport pChange; // can also inherit
 private VetoableChangeSupport vChange; // can also inherit

 public Alarm3()
 {
  alarm = -1;
  rings = 0;
  alarmTimer = new Timer("Alarm",true); // is a daemon
  pChange = new PropertyChangeSupport(this);
  pChange.addPropertyChangeListener(this);
  vChange = new VetoableChangeSupport(this);
  vChange.addVetoableChangeListener(this);
 }

 // TimerTask
```

```
 public void run()
 {
  int oldrings = rings;
  rings++;
  pChange.firePropertyChange("rings",oldrings,rings);
 }

 // PropertyChangeListener
 public void propertyChange(PropertyChangeEvent e)
 {
  System.out.println(e.getPropertyName() + " changed from "
                     + e.getOldValue() + " to " + e.getNewValue());
 }

 // VetoableChangeListener
 public void vetoableChange(PropertyChangeEvent e)
  throws PropertyVetoException
 {
//  throw new PropertyVetoException("I won't allow it.", e);
  System.out.println(e.getPropertyName() + " changed from "
                     + e.getOldValue() + " to " + e.getNewValue()
                     + " (not vetoed)");
 }

 // PropertyChangeSupport
 public void addPropertyChangeListener(PropertyChangeListener p)
 { pChange.addPropertyChangeListener(p); }

 public void removePropertyChangeListener(PropertyChangeListener p)
 { pChange.removePropertyChangeListener(p); }

 // VetoableChangeSupport
 public void addVetoableChangeListener(VetoableChangeListener v)
 { vChange.addVetoableChangeListener(v); }

 public void removeVetoableChangeListener(VetoableChangeListener v)
 { vChange.removeVetoableChangeListener(v); }

 public int getAlarm() { return alarm; }

 public void setAlarm(int newalarm) throws PropertyVetoException
 {
  vChange.fireVetoableChange("alarm",alarm,newalarm);
  alarm = newalarm;
  if(alarm < 0) alarmTimer.cancel();
```

```
  else
  {
   alarmTimer = new Timer("Alarm", true); // is a daemon
   if(alarm >= 0) alarmTimer.scheduleAtFixedRate(this,alarm,alarm);
   else alarmTimer.cancel();
  }
 }

 public int getRings() { return rings; }

 public static void main(String[] args)
 {
  Alarm3 a = new Alarm3();
  try { a.setAlarm(1000); Thread.sleep(5000); }
  catch(PropertyVetoException e)
  { System.out.println("Failed to set alarm time: " + e); }
  catch(java.lang.InterruptedException e) { }
 }
}
```

```
java ShowBean Alarm3
```

```
Info for JavaBean: Alarm3
Name: Alarm3
Description: Alarm3
Property: alarm (bound) (constrained)
Property: alarm, read: public int Alarm3.getAlarm()
Property: alarm, write: public void Alarm3.setAlarm(int)
 throws java.beans.PropertyVetoException
Property: class
Property: class, read: public final native java.lang.Class
 java.lang.Object.getClass()
Property: class, write: null
Property: rings (bound)
Property: rings, read: public int Alarm3.getRings()
Property: rings, write: null
```

14.3.10 Indexed properties

Indexed properties are array valued simple properties. Indexed properties provide additional `get` and `set` methods which take an integer index as first argument. Property changes are signalled by the method `fireIndexedPropertyChange`.

The following classes implement a bean for a Sudoku puzzle. The `Sudoku` class extends `JTable` and consequently inherits all the JavaBean properties of `JTable`.

Thus the starting Sudoku configuration can be set by editing the `model` property of `JTable`.

```
//SudokuTableModel.java

import java.lang.*;

public class SudokuTableModel
 extends javax.swing.table.DefaultTableModel
 implements java.io.Serializable
{
 private String[][] s;
 public SudokuTableModel() { s = new String[9][9]; }
 public int getColumnCount() { return 9; }
 public int getRowCount() { return 9; }
 public Object getValueAt(int x, int y) { return s[x][y]; }
 public void setValueAt(Object o, int x, int y)
 {
  s[x][y] = "";
  if(x>=0 && x<9 && y>=0 && y<9)
  {
   if((o instanceof Integer)
      && (Integer)o>0 && (Integer)o<=9)
    s[x][y] = ((Integer)o).toString();
   if((o instanceof String)
      && Integer.parseInt((String)o)>0
      && Integer.parseInt((String)o)<=9)
    s[x][y] = (String)o;
  }
 }
}

//Sudoku.java

import java.lang.*;
import java.beans.PropertyChangeEvent;
import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeSupport;

public class Sudoku extends javax.swing.JTable
 implements java.io.Serializable, javax.swing.event.TableModelListener
{
 private PropertyChangeSupport pc;

 // PropertyChangeSupport
```

```
public void addPropertyChangeListener(PropertyChangeListener p)
{
  if(pc == null) pc = new PropertyChangeSupport(this);
  pc.addPropertyChangeListener(p);
}

public void removePropertyChangeListener(PropertyChangeListener p)
{ pc.removePropertyChangeListener(p); }

public boolean getSudokuRowValid(int index)
{
 int j;
 boolean[] usedrow = new boolean[9];
 for(j=0;j<9;j++)
 {
  int v;
  try { v = Integer.parseInt((String)getValueAt(index,j)); }
  catch(Exception e) { continue; }
  if(usedrow[v-1]) return false;
  usedrow[v-1] = true;
 }
 return true;
}

public boolean[] getSudokuRowValid()
{
 int i;
 boolean[] row = new boolean[9];
 for(i=0;i<9;i++) row[i] = getSudokuRowValid(i);
 return row;
}

public boolean getSudokuValid()
{
 int i, j;
 boolean[][] usedblk = new boolean[9][9]; // check blocks
 boolean[][] usedcol = new boolean[9][9]; // check columns
 boolean[][] usedrow = new boolean[9][9]; // check rows
 for(i=0; i<9; i++)
  for(j=0; j<9; j++)
   usedblk[i][j] = usedcol[i][j] = usedrow[i][j] = false;
 for(i=0;i<9;i++)
  for(j=0;j<9;j++)
  {
   int v;
```

```
    try { v = Integer.parseInt((String)getValueAt(i,j)); }
    catch(Exception e) { continue; }
    if(usedblk[(i/3)*3+(j/3)][v-1]) return false;
    if(usedcol[j][v-1]) return false;
    if(usedrow[i][v-1]) return false;
    usedblk[(i%3)*3+(j/3)][v-1] = true;
    usedcol[j][v-1] = true;
    usedrow[i][v-1] = true;
   }
  return true;
 }

 public void tableChanged(javax.swing.event.TableModelEvent e)
 {
  int i;
  super.tableChanged(e);
  if(pc==null) pc = new PropertyChangeSupport(this);
  pc.firePropertyChange("sudokuValid",null,getSudokuValid());
  for(i=0;i<9;i++)
   pc.fireIndexedPropertyChange("sudokuRowValid",i,
       null,getSudokuRowValid(i));
 }

 public Sudoku()
 {
  super(new SudokuTableModel());
  super.setTableHeader(null);
 }
}
```

```
java ShowBean Sudoku
```

```
Info for JavaBean: Sudoku
Name: Sudoku
Description: Sudoku
...
Property: model (bound)
Property: model, read: public javax.swing.table.TableModel
 javax.swing.JTable.getModel()
Property: model, write: public void
 javax.swing.JTable.setModel(javax.swing.table.TableModel)
...
Property: sudokuRowValid (bound) (indexed)
Property: sudokuRowValid, read: public boolean[] Sudoku.getSudokuRowValid()
Property: sudokuRowValid, write: null
Property: sudokuValid (bound)
```

```
Property: sudokuValid, read: public boolean Sudoku.getSudokuValid()
Property: sudokuValid, write: null
...
```

Chapter 15

Additions to JSE 1.5 and 1.6

Java 2 Platform, Standard Edition (J2SE) version 1.5 adds a number of new features to the language. One of this addition is *generic types*. Generic types allow us to abstract over types. The most common examples are container types, such as those in the `Collection` hierarchy. The following program gives two examples. The two container classes `ArrayList` and `Hashtable` are used.

```
// GenericTypes.java

import java.util.*;

public class GenericTypes
{
   public static void main(String[] args)
   {
   ArrayList<Integer> list1 = new ArrayList<Integer>();
   list1.add(0,new Integer(-34));
   list1.add(1,new Integer(27));
   int total = list1.get(0).intValue() + list1.get(1).intValue();
   System.out.println("total = " + total);

   Hashtable<Integer,String> h = new Hashtable<Integer,String>();
   h.put(new Integer(10),"willi");
   h.put(new Integer(11),"Nela");
```

```
   Integer key = new Integer(10);
   String s = h.get(key);
   System.out.println("s = " + s);
   key = new Integer(11);
   s = h.get(key);
   System.out.println("s = " + s);
   }
}
```

We can also write our own classes with generic types. An example is given in the next program

```
// PairMain.java

class Pair<X,Y>
{
  private X first;
  private Y second;

  public Pair(X a1,Y a2)
  {
  first = a1;
  second = a2;
  }

  public X getFirst() { return first; }
  public Y getSecond() { return second; }
  public void setFirst(X arg) { first = arg; }
  public void setSecond(Y arg) { second = arg; }
  public void printPair(Pair<?,?> pair)
  {
  System.out.println("("+pair.getFirst()+","+pair.getSecond()+")");
  }
}

class PairMain
{
  public static void main(String[] args)
  {
  Pair<String,Long> l1 = new Pair<String,Long>("max",1024L);
  Pair<?,?> l2 = new Pair<String,Long>("min",64L);
  l1.printPair(l2);
  }
}
```

Java 1.5 now also has an enhanced `for` loop. The new enhanced `for` loop can replace the iterator when simply traversing through a `Collection` as follows.

```
// EnhancedFor.java

import java.util.*;

public class EnhancedFor
{
   public static void main(String[] args)
   {
   int[] a = { 1, 3, 5, 2, 4, 6, 8, 10 };
   int sum = 0;
   for(int e : a)
   {
   sum += e;
   }
   System.out.println("sum = " + sum); // => 39

   ArrayList<Double> list = new ArrayList<Double>();
   list.add(0,new Double(3.14159));
   list.add(1,new Double(2.789));
   list.add(2,new Double(4.50));
   double result = 0.0;
   for(Double d : list)
   {
   result += d.doubleValue();
   }
   System.out.println("result = " + result);
   }
}
```

Converting between basic data types such as `int`, `double`, `boolean` and their equivalent object-based counterparts like `Integer`, `Double`, `Boolean` can require unnecessary amounts of extra coding. especially if the conversion is only needed for a method call to the Collections API, for example. The autoboxing and auto-unboxing of Java basic data types produces code that is more concise and easier to follow. The following program gives an example. It simplifies the program `GenericTypes.java` given above.

```
// Autoboxing.java

import java.util.*;

public class Autoboxing
{
   public static void main(String[] args)
   {
```

```
   ArrayList<Integer> list1 = new ArrayList<Integer>();
   list1.add(0,-34);
   list1.add(1,27);
   int total = list1.get(0) + list1.get(1);
   System.out.println("total = " + total);

   Hashtable<Integer,String> h = new Hashtable<Integer,String>();
   h.put(10,"willi");
   h.put(11,"Nela");

   Integer key = new Integer(10);
   String s = h.get(key);
   System.out.println("s = " + s);
   key = new Integer(11);
   s = h.get(key);
   System.out.println("s = " + s);
   }
}
```

The following program also gives an example for autoboxing.

```
// DoubleList.java

import java.util.List;
import java.util.ArrayList;

class DoubleList
{
   public static void main(String[] args)
   {
   List<Double> doubleList = new ArrayList<Double>();

   for(double x=0.0;x<10.0;x++)
   {
   doubleList.add(x);
   }

   double sum = 0.0;

   for(double y=0.0;y<10.0;y++)
   {
   sum += y;
   }
   System.out.println("sum = " + sum);
   }
}
```

The static import feature, implemented as `import static`, enables us to refer to static constants from a class without needing to inhert from it. The following program gives an example for the `Math` class.

```
// StaticImport.java

import static java.lang.Math.*;

public class StaticImport
{
   public static void main(String[] args)
   {
   double pi = PI;
   System.out.println("pi = " + pi);
   double r = sin(PI);
   System.out.println("r = " + r);

   double d = 2.5, e = 3.0;
   System.out.println(min(d,e));
   System.out.println(PI*pow(d,e));
   }
}
```

The *varargs functionality* allows multiple arguments to be passed as parameters to methods. It requires the simple ... notation for the method that accepts the argument list and is used to implement the flexible number of arguments required for `printf`.

```
// Varargs.java

public class Varargs
{
   public static void printf(String s,Object... args)
   {
   int i=0;
   for(char c : s.toCharArray())
   {
   if(c == '%')
   System.out.print(args[i++]);
   else if(c == '\n')
   System.out.println();
   else
   System.out.print(c);
   }
   }
```

```
   public static void main(String[] args)
   {
   printf("Addition: % + % = %\n",1,1,2);
   printf("Multiplication: % * % * % = %\n",2,3,4,24);
   }
}
```

Another example for using variable arguments is given in the following summation example

```
// Varargs1.java

public class Varargs1
{
   public static int sum(int... numbers)
   {
   int total=0;
   for(int i=0;i<numbers.length;i++)
   {
   total += numbers[i];
   }
   return total;
   }

   public static void main(String[] args)
   {
   int r1 = sum(5,17);
   System.out.println("r1 = " + r1);
   int r2 = sum(18,20,-40);
   System.out.println("r2 = " + r2);
   }
}
```

The `Scanner` class provides us with a simple regular expression-oriented text scanner, for scanning primitive types and strings from an input source. The following program gives an example.

```
// ReadKeyboard.java

import java.util.Scanner;

class ReadKeyboard
{
   public static void main(String[] args)
   {
   System.out.printf("enter four values: " +
```

```
                    "integer float double string: ");
   Scanner sc = Scanner.create(System.in);
   int i = sc.nextInt();
   float f = sc.nextFloat();
   double d = sc.nextDouble();
   String s = sc.nextLine();
   System.out.printf("%d %f %4.6f %s\n",i,f,d,s);
   }
}
```

The new `java.lang.ProcessBuilder` class makes it possible to build operating system processes and manage a collection of process attributes (the command, an environment, a working directory, and whether or not the error stream is redirected). The following program gives an example. The `start()` method call starts the command executing and returns a `java.lang.Process` reference for retrieving process-exit status (in addition to other tasks).

```
// ProcessBuilding.java

import java.lang.ProcessBuilder;
import java.lang.Process;

public class ProcessBuilding
{
   public static void main(String[] args) throws java.io.IOException
   {
   Process p =
     new ProcessBuilder("notepad","c:\\javacourse\\Autoboxing.java").start();
   }
}
```

Starting from version 1.6 Java includes the ability to evaluate portions of code for scripting languages. This allows a Java program to load a custom script and execute it. The scripts can also interact with the Java program. The classes are available from the `javax.script` package. The class `ScriptEngine` implements the support for evaluating scripts from a certain language. The methods `eval(String)` and `eval(Reader)` can be used to evaluate strings in the target language or load files to be interpreted.

Not all scripting languages are supported on all platforms. To determine whether a scripting language is supported we use `ScriptEngineManager`. The `ScriptEngineManager` can find the engine for a specific language

```
ScriptEngine getEngineByName(String)
```

or list all the available engines

```
List<ScriptEngineFactory> getEngineFactories()
```

`ScriptEngineFactory` provides details about a `ScriptEngine` such as the name of the language supported and the file extensions associated with the scripting language.

The following Java program list the available scripting engines. It then attempts to evaluate any JavaScript files which are named on the command line. If the JavaScript engine is not available, then `getEngineByName` returns `null`. When trying to evaluate a script, `ScriptException` may be thrown. This indicates a problem when evaluating the script, either due to an error in the script or some other error experienced by the `ScriptEngine`.

```
// Script.java

import javax.script.ScriptEngine;
import javax.script.ScriptEngineFactory;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;
import java.util.Iterator;
import java.util.List;

public class Script
{
 public static void main(String[] args) throws Exception
 {
  ScriptEngineManager m = new ScriptEngineManager();
  Iterator<ScriptEngineFactory> i = m.getEngineFactories().iterator();
  while(i.hasNext())
  {
   ScriptEngineFactory se = i.next();
   System.out.println("Engine: " + se.getEngineName());
   System.out.println("Language: " + se.getLanguageName());
   Iterator<String> names = se.getNames().iterator();
   while(names.hasNext())
    System.out.println("  Name: " + names.next());
  }
  ScriptEngine e = m.getEngineByName("JavaScript");
  e.eval("print(\"ISSC \" + new Date() + \"\\n\");");

  int j;
  for(j=0;j<args.length;j++)
  {
   System.out.println("Running script " + args[j]);
   e.eval(new java.io.FileReader(args[j]));
  }
```

```
 }
}
```

The following file, `fact.js`, is a very simple JavaScript program.

```
function fact(n)
{
 if(n <= 0) return 1;
 return n*fact(n-1);
}
println(fact(10));
```

Running the command

```
java Script fact.js
```

yields the output

```
Engine: Mozilla Rhino
Language: ECMAScript
  Name: js
  Name: rhino
  Name: JavaScript
  Name: javascript
  Name: ECMAScript
  Name: ecmascript
ISSC Mon Aug 17 2009 13:16:53 GMT+0200 (SAST)
Running script fact.js
3628800
```

The output will differ from computer to computer and according to the Java implementation on the computer. Note that, if your version of Java does not support the JavaScript `ScriptEngine`, the program will terminate prematurely due to an exception (`NullPointerException`).

Chapter 16

Resources and Web Sites

There are a large number of web sites which provide information, news and tutorials of Java, HTML and JavaScript. Furthermore there are web sites to download new versions of Java, HTML and JavaScript.

The main web sites of SUN Microsystems for Java are

`http://java.sun.com/`
`http://java.sun.com/jdc/`
`http://developer.java.sun.com/`

Another useful web site for Java, HTML and JavaScript is "The Web Developer's Virtual Library"

`http://wdvl.com/WDVL/About.html`

The WDVL has more than 2000 pages and some 6000 links.

The web site

`http://www.niu.edu/acad/english/htmlref.html`

provides an HTML reference guide. The web site

`http://developer.netscape.com/docs/`

provides links to technical manuals and articles for Java, JavaScript and HTML.

The offical word on the latest standard in HTML can be found on the web site

`http://www.w3.org/pub/WWW/MarkUp/MarkUp.html`

Information about XML can be found on the web sites

`http://www.w3.org/XML/`

`http://www.w3schools.com/`

Bibliography

[1] Horton Ivor, *Beginning Java 2*, WROX Press, Birmingham, UK, 1999

[2] Jaworski Jamie, *Java 1.2 Unleashed*, SAMS NET, Macmillan Computer Publishing, USA, 1998

[3] Johnson Marc, *JavaScript Manual of Style*, Macmillan Computer Publishing, USA, 1996

[4] McComb Gordon, *JavaScript Sourcebook*, Wiley Computer Publishing, New York, 1996

[5] Willi-Hans Steeb, *The Nonlinear Workbook: Chaos, Fractals, Cellular Automata, Neural Networks, Genetic Algorithms, Gene Expression Programming, Support Vector Machine, Wavelets, Hidden Markov Models, Fuzzy Logic with C++, Java and SymbolicC++ Programs*, 4th edition World Scientific, Singapore, 2008

[6] Willi-Hans Steeb, Yorick Hardy, Alexandre Hardy, Ruedi Stoop, *Problems and Solutions in Scientific Computing with C++ and Java Simulations*, World Scientific, Singapore 2006

[7] Tan Kiat Shi, Willi-Hans Steeb and Yorick Hardy, *SymbolicC++: An Introduction to Computer Algebra Using Object-Oriented Programming*, 2nd edition Springer-Verlag, London, 2000

Index