Example 1. Simple text interface application (compiled and run)

```
public class HelloWorld
{
  public static void main(String argv[])
  {
    System.out.println("Hello, world!");
  }
}
```

The first line "`public class HelloWorld`" defines a "class" that is called HelloWorld. Java is a pure object oriented language, so everything has to be contained in classes (no global functions like in "C" or C++).

"`public static void main(String argv[])`" is the standard declaration for a program entry point in Java. Any class may contain a "main" function, so the same program may have several program entry points. This is useful for test and debugging purposes.

"`System.out.println("Hello, world!");`" just writes out the text that we want onto the standard output. "`System`" is a pre-defined **class** that contains the member "`out`", which is the **object** that corresponds to the standard output. "println" is a **method** of this object that takes the text to write out as a parameter. The resulting output is the given text followed by a newline character. These concepts will be explained in detail further on.

To compile this program, we first save it as a file named "`HelloWorld.java`". In fact, **the file name always has to be the same as the class name for a public class and it has to have the extension ".java"**.

Then, if we are in the same directory as the saved file, we can compile it with the DOS command "`javac HelloWorld.java`" (if the Java development environment is set up correctly).

We run it with the command line "`java HelloWorld`" (if we are still in the same directory and the Java environment is set up correctly).

2. Java development tools

[redacted]All the basic tools needed are still **free**.

General information about Java and the corresponding development tools may be found at the address http://www.javasoft.com.

2.1 The Java Development Kit (JDK)

The **Java Development Kit (JDK)** is the minimal file you need to download in order to develop in Java. The version that is meant for the Windows environment contains an automatic installer that takes care of the installation. Only setting up the Windows PATH may require some manual intervention.

The JDK contains everything you need to develop general-purpose programs in Java:

- **Base Tools**
  - **javac**: The Java Language Compiler that you use to compile programs written in the Java(tm) Programming Language into bytecodes.
  - **java**: The Java Interpreter that you use to run programs written in the Java(tm) Programming Language.
  - **jre**: The Java Runtime Interpreter that you can use to run Java applications. The jre tool is similar to the java tool, but is intended primarily for end users who do not require all the development-related options available with the java tool.
  - **jdb**: The Java Language Debugger that helps you find and fix bugs in Java(tm) programs.
  - **javah**: Creates C header files and C stub files for a Java(tm) class. These files provide the connective glue that allow your code written in the Java Programming Language to interact with code written in other languages like C.
  - **javap**: Disassembles compiled Java(tm) files and prints out a representation of the Java bytecodes.
  - **javadoc**: Generates API documentation in HTML format from Java(tm) source code. Also see Writing Doc Comments for Javadoc.
  - **appletviewer**: Allows you to run applets without a web browser.

- **RMI Tools**
  - **rmic**: Generates stub and skeleton class files for Java objects implementing the java.rmi.Remote interface.
  - **rmiregistry**: Starts a remote object registry on a specified port. The remote object registry is a bootstrap naming service which is used by RMI servers.
  - **serialver**: Returns the serialVersionUID for one or more classes.
- **Internationalization Tools**
  - **native2ascii**: Converts non-Unicode Latin-1 (source code or property) files to Unicode Latin-1.
- **JAR Tool**
  - **jar**: Combines multiple files into a single Java Archive (JAR) file.
- **Digital Signing Tool**
  - **javakey**: Generates digital signatures for archive files and manages the database of entities and their keys and signatures.
- **Environment Variables**
  - **CLASSPATH**: Tells the Java Virtual Machine and other Java applications where to find the class libraries.

If you have developed an application in Java and want to deploy it, you need to download the **Java Runtime Environment (JRE)**, which contains only what is necessary for running a Java application (JVM and libraries), but no development tools. It may be freely distributed with your application.

It is possible to use the JDK as the only development environment for any development project. It is, however, useful to create straight away at least the following two script files for each project:

- **makeit.bat**: Contains the command line for compiling the whole project. It may be useful to associate this with using various "make" tools for project management.
- **runit.bat**: Launches the program.

3. Java Syntax

Java syntax is very close to that of "C" and C++. This is true for

- Comments (+ documentation comment).
- Variable names, declarations and initialisations.
- Most keywords, with a few new ones.
- Control flow ("if", "while", "for", ...)
- etc.

3.1 Variable declarations

Standard primitive types in Java:

boolean	true or false
char	16 bit character, coded using Unicode 1.1.5
byte	8 bit signed integer, using 2's complement
short	16 bit signed integer, using 2's complement
int	32 bit signed integer, using 2's complement
long	64 bit signed integer, using 2's complement
float	floating point real number, 32 bit IEEE 754-1985
double	floating point real number, 64 bit IEEE 754-1985

Variable declarations consist of three parts: **modifiers**, followed by a **type**, followed by a **list of identifiers**. The modifiers are optional, but the type and list of identifiers is not.

Example: `public int   a, b, c;  // "public" is a modifier.`

Variables may be initialised as in "C", e.g. "`int a = 0;`"

3.2 Arrays

Arrays are NOT declared as in "C". To declare an array of integers, we would write "`int[] ia;`". This means that `ia` is a **reference** (compare with reference and pointer in "C" and C++) to an array of integers.

In order to actually create the array at the same time, we would write:

```
int[]    ia = new int[3];
```

This means that `ia` now references an array object which can contain an array of 3 integers.

Example 5. Table initialisation and index exceptions (compiled and run)

```
public class TblExample
{
  public static void main(String argv[])
  {
    int  i;
    int[]  itab1 = {10, 20, 30};   // Table initialisation.

    for ( i = 0 ; i < itab1.length ; i++ )
      System.out.println(itab1[i]);

    // Create an "IndexOutOfBoundsException" here.
    for ( i = 0 ; i <= itab1.length ; i++ )
      System.out.println(itab1[i]);
    System.out.println("We got away from the Exception!");
  }
}
```

Screen output:

```
10
20
30
10
20
30
java.lang.ArrayIndexOutOfBoundsException
        at TblExample.main(Compiled Code)
```

Example 5 shows how tables can be initialized in Java. A VERY IMPORTANT improvement in Java compared to languages like "C" and C++ is that tables "know" themselves how big they are. This is why we can get the length of the table with "`itabl1.length`".

Just like in "C", table indexes start from 0. That is why the second loop goes too far.

Invalid table indexes are automatically detected in Java. When this occurs, we say that an exception occurred. In programming terms we call this **throwing an exception**.

If an exception occurs in a program that does not handle it, then the program exits immediately. If our program wants to handle exceptions graciously, we could define an **exception handler** by replacing the for-loop that provokes the exception with the following code:

```
// Create an "IndexOutOfBoundsException" here.
try {
   for ( i = 0 ; i <= itab1.length ; i++ )
      System.out.println(itab1[i]);
}
catch ( IndexOutOfBoundsException e ) {
       System.out.println("We caught it!");
}
```

There are a lot of pre-defined exception classes in Java and the programmer may define new ones. This will be treated more in detail later, but this is a very important feature in the Java language, **which makes Java programs much more stable than "C" or C++ programs**, for instance.

3.3 Flow control

Flow controls which have an identical syntax to that of "C" are:

- if-else
- switch-case
- while
- do-while
- for
- break, continue, return

But there is no "`goto`" statement! Instead, Java proposes labelled breaks, as shown in the following example.

Example 6. Labelled breaks (compiled and run)

```
public class LabelledBreak
{
   public static void main(String argv[])
   {
brkpnt:for ( int i = 0 ; i < 10 ; i++ ) {
         for ( int j = 0 ; j < 10 ; j++ ) {
            if ( j == 3 ) break brkpnt;
            System.out.println("i = " + i + ", j = " + j);
         }
      }
   }
}
```

Screen output:

```
i = 0, j = 0
i = 0, j = 1
i = 0, j = 2
```

If we would have an ordinary "`break;`" in Example 6, then we would go through the outer loop 10 times and through the inner one 3 times for each outer loop repetition. However, the labelled break breaks the outer loop straight away and we do not even finish the first repetition of the outer loop.

3.4 Standard input and output

The global class "System" contains the fields "in", "out" and "err", which correspond to standard input, standard output and standard error. Using these is illustrated by the following example:

Example 7. Using standard input and output in Java (compiled and run)

```
import java.io.*;

/**
 * Example showing the use of standard input and output.
 */
class StdioExample
{
   public static void main(String args[]) throws IOException
   {
      int n;

      BufferedReader stdin =
         new BufferedReader(new InputStreamReader(System.in));

      System.out.print("Enter an integer: ");
//    System.out.flush();
      n = Integer.parseInt(stdin.readLine());

      System.out.println("Square: " + n*n + "  Cube: " + n*n*n);
   }
}
```

Program output:

```
Enter an integer: 9
Square: 81  Cube: 729
```

"`System.in`" is an object of class "`InputStream`". This class offers very basic reading capabilities, which is the reason why we "embed" it into an "`InputStreamReader`" and then a "`BufferedReader`" (more about this in the chapter about file reading and writing).

The commented line "`System.out.flush();`" might be necessary on some systems, where the standard output is buffered. In that case the display might be deferred and not occur until the buffer is full.

"`n = Integer.parseInt(stdin.readLine());`" is one possibility to convert a string into an integer. See what happens if you do not feed in an integer!

`Integer` is a standard Java class, which contains a (static) member function called "`parseInt()`" that converts the given argument string into an integer and returns the value.

"`System.out.println()`" always flushes the output stream automatically.

"`throws IOException`" in the method declaration means that the method might throw an exception of type `IOException`. This exception might occur in the call to "`readLine()`" and since we do not catch it, we just pass it on.

3.5 Programming style.

It is highly recommended to use the same programming conventions that have been used by Sun in their class libraries. This means:

- Indentation.
- Class names (ThisIsAClassName).
- Constant names (THIS_IS_A_CONSTANT).
- Class variable/field names (thisIsAClassMemberVariable).
- Method names (thisIsAMethodName).
- Function parameter names (thisIsAFunctionParameter).
- Local variable names (this_is_a_local_variable). This does not seem to be used by Sun, but is highly recommended to increase program readability.
- Comments (class and method definitions, documentation comments).

4. Classes and Objects

A class is a template that defines how an object will look and behave once instantiated.

In this course, we will use cars as an example of a system with various objects that interact with each other. A car contains subsystems which may be considered as objects, such as:

- Injection system.
- Ignition system.
- Breaking system (including ABS).
- Steering system.
- Others (have a look at your car's fuse boxes to get an idea).

A **class** corresponds to the factory or machine that produces these subsystems. Once these subsystems are installed in a car, they become **objects** that communicate through well-defined protocols in such a way that (hopefully) makes the car work and makes it possible to drive with it.

In Object Oriented Programming (OOP), **methods** implement the communication protocol between the objects.

Only some of all the methods are visible to other objects. These are the **public methods**.

We also have **private methods** and **private variables**, which are only visible to the object itself. These are used for the internal operations of the object.

This is called **encapsulation** in OOP. In a car, it means that there is no way for one subsystem to interfere with the internal functioning of another one. This is natural, since there is no way why the injection system should have access to the private calculations of the ABS system, for instance.

We may also have **private classes**. The ABS system might, for instance, be a private class to the breaking subsystem. It would then be directly accessible only to the breaking subsystem.

Another important concept in OOP is **inheritance**. This means that we can define **class hierarchies**, where **derived classes** inherit behaviour from their **base class**. In a car, we could for instance say that the clutch, break and accelerator classes are all derived from the "PedalControlledDevice" base class. Then the "PedalControlledDevice" would define all the behaviour common to the derived classes.

The third classical concept in OOP is **polymorphism**, which means that we may have several methods which the same name, but with different parameters. This means that the actual behavior of the method depends on the kind of data that it works on. An example of this is the "toString()" method that is defined for all classes in Java and gives a textual description of the object, no matter what the type of the object is.

4.1 Class declarations

A class declaration has the following syntax:

```
[public, private or protected] class <classname> [extends <baseclassname>]
[implements <classname, classname, …>]
{
  /* Variable and method declarations. */
}
```

Example: A minimal class definition for a bank account.

```
public class BankAccount
{
} // No semicolon
```

A minimal class declaration just contains the keyword "`class`" and the classname. A class is private by default, which is usually not very useful. If it should not be private, the class declaration should be preceded either by the keyword "`public`" or the keyword "`protected`".

A public class is visible in the entire program. A protected class is accessible only to classes in the same **package** (more about packages later). In both cases, the class definition has to be in a file of its' own, which has the same name as the class (**case sensitive!**) and the extension "`.java`".

If the class is derived from another class, then we add "`extends <baseclassname>`" to indicate the base class to use. There may be only one base class.

If there is no "`extends`" statement, then the class has the "`Object`" class as its' base class. In Java, the class "`Object`" is the root of the class hierarchy, so it is the ultimate base class for all classes.

The "implements" keyword is a way of achieving a kind of multiple inheritance. It is possible to implement several **abstract classes** or **interfaces** with this keyword. Abstract classes and interfaces will be treated later, but C++ programmers should at least know what abstract classes mean.

Attention! There is no semicolon after the closing bracket in Java, which is a difference from "C" structures and C++ classes.

4.2 Class variables and instance variables

Every object stores its' state in its' **instance variables**. For a bank account, instance variables could be "`balance`" and "`id`". There may also be instant variables that are set to constant values (like "`#define`" or "`const`" in "C").

Example: Class definition with instance variables and constants.

```
public class BankAccount
{
   // Class variable
   public static int nextID = 100;

   // Instance variables
   private doublebalance;
   private int    id;

   // Constant value
   private final int  PREFERENCE_CLIENT_LIMIT = 10000;
}
```

All variable declarations are private unless specified otherwise (so the "`private`" keywords in the example are not necessary).

The keyword "final" signifies that we define a constant value.

Public variables are accessible from any other object in the application. Protected variables are only accessible to the classes in the same class package.

Class variables are indicated with the keyword "`static`". Class variables can be used directly through the class name, e.g. "`BankAccount.nextID`". So, this is a property of the class itself, not a property of the objects of this class (it is, however, directly accessible to instance methods).

Usually static variables are initialised at their point of declaration, but a class may also have a special **static initialisation block**.

Example: Static initialisation block.

```
static {
  nextID = 0;
}
```

4.3 Method definitions

Methods are functions that are associated to a class (**class methods**) or the objects of a class (**instance methods**).

There is a special kind of method that is called a **constructor method**. This is a method that is called when a new object is created using the "`new`" keyword. A constructor initialises the newly created object, which mainly means that it sets up values for the object's instance variables. There may be several constructors in the same class.

Example: Constructors for the "BankAccount" class.

```
  public BankAccount() { this(0.0) ; }

  public BankAccount( double initBal )
    { balance = initBal; id = newID(); }
```

The name of a constructor function is always the same as the class name and there is no return type, but it should be declared public.

There are two constructors for the class "BankAccount". The first one is the default one, which takes no parameters. This one is used in a call like "`BankAccount ba = new BankAccount()`". The only thing that it does is to call the other constructor, which takes an initial balance as a parameter.

The second constructor sets the balance of the new "BankAccount" object to the value passed as a parameter and affects it a new ID.

Normal method declarations consist of an access specifier ("`public`", "`protected`" or "`private`"), a return type, the name of the method and the list of parameters and their type. There may be class methods and instance methods.

Access to methods is determined in the same way as for variables.

"`newID`" is a class method that returns the next unused account ID. This method has to be a class method, since there is no way that the newly created object could know what ID to use by itself. It is like in a car factory – it is the factory that knows what serial number to give to a newly manufactured car, not the car itself.

Example: The class method newID.

```
private static int newID() { return nextID++; }
```

A class method may be accessed from an object of another class (if it is not private) simply by writing the class name, a dot and the name of the method (e.g. "`BankAccount.newID`" if "`newID`" would not be private).

Just like instance variables, instance methods can only be accessed through an object. So, if we have an object "`ba`" of class "`BankAccount`", we could access the method "`balance`" by writing "`ba.balance()`".

Example: Instance methods of "BankAccount".

```
public double balance() { return balance; }
public int    id()       { return id; }

public void withdraw(double amt) { balance -=amt; }
public void  deposit(double amt) { balance +=amt; }

public String toString()
  { return super.toString() + "(id:" + id + ", bal:" +
    balance + ")" ; }
```

4.4 Creating and referencing objects

What happens in Java when we use the operator "new" is that the virtual machine reserves enough memory for containing all the data of the newly created object. The result of the operation is a reference to the newly created object.

In Java, all objects are situated in "global memory". This means that we do not need to worry about stack handling and "copy constructors" in the same way as in C++.

Now, let us have a look at a small example that shows how objects are created and how references to them are managed:

Example: Object creation and referencing (compiled).

```
public class ObjCreationExample
{
  public ObjCreationExample   firstObject, secondObject;

  public static void main(String args[])
      {
    ObjCreationExample object1 = new ObjCreationExample();
    ObjCreationExample object2 = new ObjCreationExample();
    ObjCreationExample object3 = new ObjCreationExample();
    ObjCreationExample object4 = new ObjCreationExample();
    ObjCreationExample object5 = new ObjCreationExample();
    ObjCreationExample object6 = new ObjCreationExample();
    object1.firstObject = object3;
    object1.secondObject = object5;
    object2.firstObject = object4;
  }
}
```

This example sets up the "object space" shown in Figure 4, which shows the objects that we created and the reference links that we set up. In fact, what we have set up is a minimal kind of "linked list" (quite uncompleted here), where one object has a link to the next one and so on.

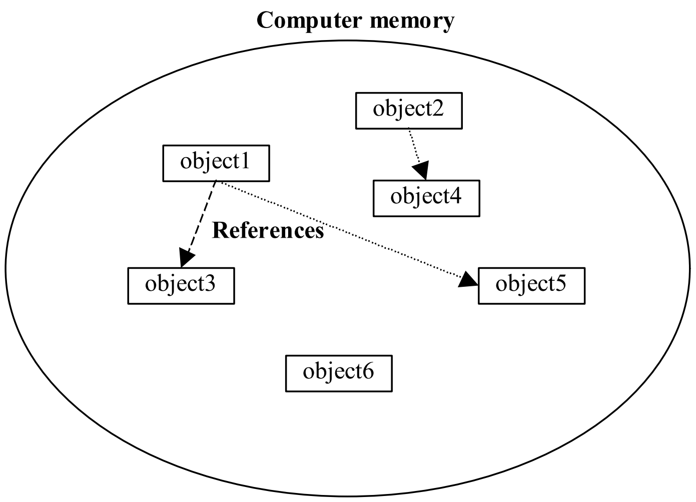

Figure 4. Illustration of "object space".

Java contains an operator which allows us to check the class that an object belongs to, "`instanceof`". So, the test "`object1 instanceof ObjCreationExample`" is true in the preceding example, while "`object1 instanceof Point`" is false.

"`instanceof`" also works for base classes. This means that the test "`object1 instanceof Object`" is true too.

All objects are uniquely identified by their address. The memory address is different if you have different pieces of data.

4.5 Garbage collection

In Java, there is no "`delete`" keyword. It is not necessary thanks to **garbage collection**.

Garbage collection means that there is an automatic functionality that regularly goes through the object space and checks which objects are still referenced and which are not. Those which are no longer referenced are unusable for the program, so it is safe to liberate the space they use.

In the example of the previous chapter, all the objects would be automatically deleted once we exit from the "main" function (well, they would anyway since we exit the program). This is because all the variables that reference them are local to the main-function, so they no longer exist once we return from main. And since these references no longer exist, there is no reason to keep all these objects in memory.

Having no "`delete`" operator, there is no destructor function in Java neither (unlike C++). However, if there are any non-Java resources used by the object that have to be liberated (a communications port reserved by the object, for instance), there may be a "`finalize()`" method, that is called before the object is destroyed by garbage collection.

Garbage collection was also used in Smalltalk from the start.

4.6 Java Bank Account

Now our complete "BankAccount" example looks like this:

Example 8. Complete BankAccount example (compiled and run)

```
public class BankAccount
{
   // Class variable
   public static int nextID = 100;

   // Instance variables
   private doublebalance;
   private int    id;

   // Constant value
   private final int  PREFERENCE_CLIENT_LIMIT = 10000;

   public BankAccount () { this(0.0) ; }

   public BankAccount( double initBal )
      { balance = initBal; id = newID(); }

   public double balance() { return balance; }
   public int    id()       { return id; }

   public void withdraw(double amt) { balance -=amt; }
   public void  deposit(double amt) { balance +=amt; }

   public String toString()
      { return super.toString() + "(id:" + id + ", bal:" +
        balance + ")" ; }

   // Class method
   private static int newID() { return nextID++; }

   // Another "special" class method
   public static void main(String args[]) {

      BankAccount a=new BankAccount(15.25);
      BankAccount b=new BankAccount();

      System.out.println("a=" + a.toString() );
      System.out.println("b=" + b.toString() );

      a.withdraw(5.50);
      b.deposit(125.99);

      System.out.println("a=" + a);
      System.out.println("b=" + b);

      System.exit(0);
   }
} // no semi-colon
```

Screen output:

```
a=BankAccount@1cc803(id:100, bal:15.25)
b=BankAccount@1cc802(id:101, bal:0.0)
a=BankAccount@1cc803(id:100, bal:9.75)
b=BankAccount@1cc802(id:101, bal:125.99)
```

The method "`toString()`" is special since it allows us to make a string value of the object.

This name is fixed; you have to use that name. It is the function that is used in the operation "`System.out.println("a=" + a);`" that writes out a string describing the object.

It cannot take any parameters and it must return a String value.

If we look a little closer at the toString method, we notice something like "`super.toString`". The word "`super`" is a keyword in the Java language, and it says that the super class' "`toString`" is going to be invoked.

Since we have not defined any superclass for "`BankAccount`", the class "`Object`" is implicitly used as base class.

The "`toString`" method of class "`Object`" actually returns the class name, @, and then the address of the object. You don't have to use super.toString if you don't want to; it's there only if you need it. This also guarantees that you can use the string concatenation operator "+" for all objects.

At the end, the "System.exit(0)" statement means to stop the Java virtual machine right now with an exit value of 0. This statement is not necessary here, in fact, but it is very useful for exiting the program in severe error situations.

4.7 Class Hierarchies and Inheritance

When developing an object oriented program, it is essential to use existing class libraries as much as possible, since it reduces the programming work needed and simplifies maintenance.

Example 9. Circle drawing using a class derived from the standard "Point" class (compiled and run).

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

public class CircleExample extends Applet
{
  private final int  RADIUS_INCREMENT = 6;

  private MyPoint  myPoint1, myPoint2, myPoint3;

  public static void main(String argv[])
  {
    Frame  f;

    f = new Frame();
    f.setLayout(new GridLayout());
    f.add(new CircleExample());
    f.setSize(250, 200);
    f.show();
  }

  CircleExample()
  {
    myPoint1 = new MyPoint(20);
    myPoint2 = new MyPoint(50, 50, 10);
  }

  public void paint(Graphics g)
  {
    myPoint1.drawIt(g);
    myPoint2.setLocation(50, 50);
    myPoint2.drawIt(g);

    // Then we use some information about our size to draw a
      // centered figure.
    Dimension d = size();
    int r = 1;
            myPoint3 = new MyPoint(d.width/2, d.height/2, r);
      myPoint3.setFilled(false);
            while ( r <= Math.sqrt(d.width*d.width + d.height*d.height) ) {
      myPoint3.drawIt(g);
      r += RADIUS_INCREMENT;
      myPoint3.setRadius(r);
    }
  }
}

class MyPoint extends Point
{
  private int      radius;
      private boolean   isFilled = true;

  MyPoint(int r)
      {
    radius = r;
  }

  MyPoint(int x, int y, int r)
  {
    super(x, y);
    radius = r;
  }

  public void setRadius(int r)
  {
    radius = r;
    }

  public void drawIt(Graphics g)
    {
    if ( isFilled )
      g.fillOval(x - radius, y - radius, 2*radius, 2*radius);
    else
      g.drawOval(x - radius, y - radius, 2*radius, 2*radius);
    }

  public void setFilled(boolean b)
  {
      isFilled = b;
    }
```

```
}
```

This program gives the window shown in Figure 5.

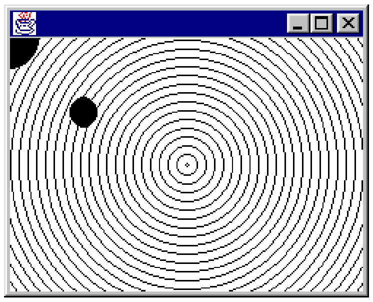

Figure 5. Window of the CircleExample program.

In Example 9 we have used the standard class "`Point`" as a base class for our own class "`MyPoint`" (it should actually have been called "Circle" or simething alike).

This means that "MyPoint" inherits some useful properties and methods, like a constructor for setting the initial position and a "`setLocation()`" method for changing it.

Also notice that "MyPoint" is a private class. This is sufficient since it is a class that is used only by the class "`CircleExample`" and since it is defined in the same source file as "CircleExample".

4.8 How to declare and use global constants and variables

The only things that are "global" in Java are classes. So it is impossible to declare global constants or variables anywhere you like as you would in "C" or C++.

Since having global constants and variables is very useful in most industry-scale applications, there is a very convenient workaround for the problem.

One solution is to define a few special classes, which contain these constant declarations and variables as static members.

It might be a good solution to have one class for the global constants, one for the global variables and one for general utility functions.

Example: Classes for global constants, variables and functions.

```
public class GlobalConstants
{
  public static int  A_GLOBAL_CONSTANT = 10;
  ...
}

public class GlobalVariables
{
  public static int  aGlobalVariable;
  ...
}

public class GlobalFunctions
{
  public static void  aGlobalFunction(int val1, int val2) {}
  ...
}
```

Now these are accessible from anywhere in the program as "`GlobalConstants. A_GLOBAL_CONSTANT`", "`GlobalVariables.aGlobalVariable`" and "`GlobalFunctions.aGlobalFunction(...)`".

This convention requires some more writing for referencing global items, but it has the advantage of clearly indicating when global items are used and modified. This is extremely important for program documentation purposes and for program maintainability.