INRTRODUCTION TO OBJECT – ORIENTED PROGRAMMING WITH JAVA

Table of Contents

RCS 102: OBJECT ORIENTED PROGRAMMING I 3 (2L, 1P) Units

Prerequisite: RCS 100

Objectives

The course first aims to introduce to the students the object oriented (OO) programming paradigm and then to teach them how to write structured and object-oriented programs in Java. This will highlight the use of control structures, the use of language Libraries and other tools.

Learning outcomes:

At the end of the module, the student

1. should have a working knowledge of a widely used object oriented programming language
2. be able to design, code, debug and document object oriented programs.

Course Contents

Introduction. Short Java history, program structure, Edit-Compile-Run cycle, Object Oriented Concepts, data types, variable declaration, constants, assignment statements, operators, string manipulation (24%)

Input & Output. Standard output & Input, other example of standard classes e.g. Math class, Gregorian calendar class (5%)

Defining your own classes. Instantiable class, constructors, visibility modifiers, local variables, return values, parameter passing, accessors, mutators, organizing classes into packages, making an instantiable class the Main class. (20%)

Control Structures. Algorithms, Stepwise refinements, Decisions (if, if-else, switch), Repetitions (for loop, while loop, do..while loop), The Break statement and the continue statement (13%)

Arrays. Array basics, multidimensional arrays, arrays of objects, passing arrays to methods (9%)

Inheritance and Polymorphism. Inheritance, Method overloading and overriding, Polymorphism, more modifiers: protected & default, static & final, static methods & variables, abstract classes and interfaces (13%)

Exception handling. Catching exceptions, throwing exceptions, propagating exceptions, types of exceptions, programmer-defined exceptions. (7%)

File Manipulation. File concepts, uses of files, text and binary files, sequential and random file access, Processing a text file, Random Access Files. (9%)

LAB WORK: The lab sessions should be used to test student programs (20 percent).

READINGS

1. Mark & Allen & Weiss, Data Structures and Problem solving using Java
2. P.A. Lee & C. Phillips, The Apprentice C++ Programmer
3. Glenn W. Rowe, An Introduction to Data Structures and Algorithms with Java

Chapter 1: INTRODUCTION

1. 1 Short History of Java

Java was conceived to develop advanced software for a wide variety of network devices and systems. The language drew from a variety of languages such as C++, Eiffel, SmallTalk, Objective C. The result is a language platform that has proven suitable for developing secure, distributed, network-based end-user applications in environments ranging from network embedded devices to the World-Wide Web and the desktop. Java attempts to provide a platform for the development of secure, high performance, robust applications on multiple platforms in heterogeneous, distributed networks. Java does that by being architecture neutral, portable, and dynamically adaptable.

Task:

If you are interested Read more on History of Java from these pages:

- http://www.oracle.com/technetwork/java/javase/overview/javahistory-index-198355.html
- http://www.oracle.com/technetwork/java/javase/overview/javahistory-timeline-198369.html
- http://ei.cs.vt.edu/book/chap1/java_hist.html
- http://www.particle.kth.se/~lindsey/JavaCourse/Book/Part1/Java/Chapter01/history.html
- Search for Java History from Google or any of your favorite search engine.

1.2 Characteristics of Java

1.2.1 Java Programming Language, Java Virtual Machine and Java Platform

These three terms are somewhat confusing but are key to understanding Java. The Java programming language is the language in which Java applications (including applets, servlets, and JavaBeans components) are written. It is a state-of-the-art, object-oriented language that has a syntax similar to that of C. It is a powerful language but avoids the overly complex features that have bogged down other object-oriented languages, such as C++. You can write robust, bug-free code using Java as compared to other Object Oriented Programming.

When a Java program is compiled, it is converted to byte codes that are the portable machine language of a CPU architecture known as the Java Virtual Machine (also called the Java VM or JVM). The JVM can be implemented directly in hardware, but it is usually implemented in the form of a software program that interprets and executes byte codes.

The Java platform is distinct from both the Java language and Java VM. The Java platform is the predefined set of Java classes that exist on every Java installation; these classes are available for use by all Java programs. Java classes are organized into related groups known as packages. The Java platform defines packages for functionality such as input/output, networking, graphics, user-inter face creation, security, and much more. The Java platform is also sometimes referred to as the Java runtime environment or the core Java APIs (application programming interfaces).

1.2.2 Characteristics

- Simple: The fundamentals are learned quickly; programmers can be productive from the very beginning.
- Familiar: Java looks like a familiar language C++, but removes some of the complexities of C++
- Object oriented: so it can take advantage of modern software development methodologies. Programmers can access existing libraries of tested objects, which can be extended to provide new behavior.
- Portable: Java is designed to support applications capable of executing on a variety of hardware architectures and operating systems. The architecture-neutral and portable language platform of Java is known as the Java Virtual Machine
- Multithreaded: for applications with many concurrent threads of activity.
- Interpreted: for maximum portability and dynamic capabilities.
- Robust and Secure: Java provides extensive compile-time and run-time checking. There are no explicit pointers, and the automatic garbage collection eliminates many programming errors. Sophisticated security features have been designed into the language and run-time system.

1.2.3 Edit-Compile-Run

To write and edit Java source code you need to have a text editor, such as the "NotePad" editor or many other simple text editor including gedit, vi, vim, emacs, notepad++, eclipse, netbeans, Textpad, etc. In this course we will use NetBeans, the text editor produced by Sun Microsystems. It is very powerful and will make Java very easy to you. Unfortunately, it will make you very lazy. So better start with notepad, then later shift to NetBeans.
Java has certain requirements about how you must save files which contain Java code if you are going to make use of them. In particular, each file contains the definition for one class, and each file must be saved under the name classname.java, where classname is the name of the class which that file defines.

Before you can do anything with your Java code, you have to compile it; "compiling" refers to the process by which high-level, human-readable Java code is turned into a long string of bytes (much like a machine language) which can be understood by a Java interpreter.

After you have written your code and are ready to compile it into the "byte code", you should save your work from your editor. Be sure to follow Java file-naming conventions and save your work onto the machine's hard drive.

If your program compiled successfully, it should have created a new file; this new file will have the same name as the old one, except that it will end in ".class" rather than ".java". This file contains the byte code version of the Java code you wrote.

In summary: The basic steps to develop a Java program are:

1. Use an editor to edit source files.
2. Save the file. The file name should be the same as the name of the class containing ***main*** function.
3. Use Java compiler, ***javac***, to translate the source file into binary format, which can be read or executed (run) by a computer.
4. If there are error message, repeat step 1 to 3.

5. Run the executable file by Java interpreter, ***java***, to check the results.
6. Repeat steps 1-4 until a satisfactory program is developed.

Read More

- NetBeans use slightly different procedures, read on how to use it from this site: http://netbeans.org/kb/docs/java/quickstart.html

1.3 Components of a java Program

The following example shows the basic components of a Java program:

```
// Our first Java program
// Written on 15/03/2011
// File: HelloWorld.java
// The famous Hello World!

public class HelloWorld {
   public static void main() {
      System.out.println("Hello World");
   }
}
```

The program starts with a comment:

```
// Our first Java program
// Written on 15/03/2011
// File: HelloWorld.java
// The famous Hello World!
```

- all the characters after the symbols `//` up to the end of the line are ignored;
- they do not change the way the program runs, but they can be very useful in making the program easier to understand;
- they should be used to clarify some part of a program, to explain what a program does and how it does it.

There could also be comments beginning with a `/*` and continue, possibly across many lines, until a `*/` is found, like so:

```
/*
   This is
   a comment that continues
   across lines
*/


/*================
=  This is another
=  comment that continues
=  across lines
==================*/
```

Other components of this first program:

- A class definition: Java programs include at least a class definition such as `public class HelloWorld`. The class extends from the first opening curly brace `{` to the last closing curly brace `}`
- The `main()` method: a method is a collection of programming statements that have been given a name and that execute when called to run. Methods are also delimited by curly braces.

The `main()` method

- all Java applications (not applets) must have a class (only one) with a `main()` method where execution begins;
- programming statements within `main()` are executed one by one, until its termination;
- the `main()` method is preceded by the words `public static void` called modifiers;
- the `main()` method always has a list of command line arguments that are passed to the program `main(String[] args)` (which we are going to ignore for now)

Statements

Statements are

- instructions to the computer to determines what to do
- end with a semicolon `';'` (a terminator, not a separator)

In this example there is only one statement `System.out.println("Hello world");` to print a message and move the cursor to the next line, by using the method `System.out.println()` to print a constant string of characters and a newline. Within a method the statements are executed in sequential order: sequential execution.

Reserved words

`class, static, public, void` have all been reserved by the designers and they can't be used with any other meaning. The following is a list of reserved words in Java:

abstract	default	if	package	synchronized
assert	Do	implements	private	this
boolean	double	import	protected	throw
break	else	instanceof	public	throws
byte	extends	int	return	transient
case	f a l s e	interface	short	t r u e
catch	final	long	static	Try
char	finally	native	strictfp	void

class	float	new	super	volatile
const	for	null	switch	while
continue	goto			

Case sensitive

Java compilers are case sensitive, meaning that they see lower case and upper case differently. Upper / lower case should be used so programmers can better read the code. Any literal used for identification of an entity is called an identifier. In Java, the identifiers `aString AString ASTRING` are all different:

1.4 Programming Style

Adhering to a style makes programs easier to read for humans. There are rules that most Java programmers follow, such as:

- One statement per line
- Indentation: 3 spaces. Indicates dependency between statements.
- Comments should be added to clarify code sections that are not obvious
- Blank lines: should be used to separate different logic sections of the code
- Naming (identifiers): any combination of letters, digits, dollar signs `'$'` underscore characters `'_'`, not beginning with a digit, is legal. Identifiers should be meaningful, but not verbose:
    - Legal: `Total`, `lastWord`, `TaxCalculation`
    - Illegal: `3rdAmmendment`, `you too`, `you#too`
    - Wrong: `s1`, `theFirstOfTheStudentsInTheClass`
    - Right: `student1`, `stud_1`, `firstStudent`

1.5 Programming Errors

Programming errors can be divided into:

- Compilation errors: are detected by the compiler at compilation time. The executable is not created.
- Execution errors: appear when the program runs. Typically execution stops when an exception such as this happens, it is possible to handle these errors
- Logical errors: the program compiles and runs with no problems but gives out wrong results

Programs should be carefully tested and debugged so the problems are corrected.

1.6 Java Data Types

Java has two major data types, these are Primitive Data types and Objects. Primitives (such as int) are

created and manipulated directly using its name. On the other hand Objects are handled through references. Objects will be covered later in this chapter. Table below summarizes the primitive types.

Type	Contains	Default value	Size	Range
boolean	true or false	f a l s e	1 bit	NA
char	Unicode character	\u0000	16 bits	\u0000 to \uFFFF
byte	Signed integer	0	8 bits	–128 to 127
short	Signed integer	0	16 bits	–32768 to 32767
int	Signed integer	0	32 bits	–2147483648 to 2147483647
long	Signed integer	0	64 bits	–9223372036854775808 to 9223372036854775807
float	IEEE 754 floating point	o.o	32 bits	±1.4E–45 to ±3.4028235E+38
double	IEEE 754 floating point	o.o	64 bits	±4.9E–324 to ±1.7976931348623157E+308

The boolean Type

The boolean type represents truth values. There are only two possible values of this type, representing the two boolean states: on or off, yes or no, true or false. Java reserves the words true and false to represent these two boolean values.

The char Type

The char type represents Unicode characters. To include a character literal in a Java program, simply place it between single quotes (apostrophes):

```
char c = 'A';
```

You can, of course, use any Unicode character as a character literal, and you can use the \u Unicode escape sequence. In addition, Java supports a number of other escape sequences that make it easy both to represent commonly used nonprinting ASCII characters such as newline and to escape certain punctuation characters that have special meaning in Java. For example:

```
char tab = '\t', apostrophe = '\'', nul = '', aleph='\u05D0';
```

The table lists the escape characters that can be used in char literals. These characters can also be used in string literals, which are covered later in this chapter.

Java Escape Characters	
\b	Backspace
\t	Horizontal tab
\n	Newline
\f	Form feed

\r	Carriage return
\”	Double quote
\'	Single quote
\\	Backslash
\xxx	The Latin-1 character with the encoding xxx, where xxx is an octal (base 8) number between 000 and 377. The forms \x and \xx are also legal, as in '\0', but are not recommended because they can cause difficulties in string constants where the escape sequence is followed by a regular digit.
\uxxxx	The Unicode character with encoding xxxx, where xxxx is four hexadecimal digits. Unicode escapes can appear anywhere in a Java program, not only in character and string literals.

The char type can be converted to a number of integral type. The Character class defines a number of useful static methods for working with characters, including isDigit(), isJavaLetter(), isLowerCase(), and toUpperCase().

Integer Types

The integer types in Java are byte, short, int, and long. These four types differ only in the number of bits and, therefore, in the range of numbers each type can represent. All integral types represent signed numbers; there is no unsigned keyword as there is in C and C++.

You can represent the integer literals in octal or in hexadecimal. The literal that begins with 0x or 0X is an hexadecimal number (allowing a to f or A to F) and literal that starts with leading zero (0) is an octal number (disallowing 8 and 9). Examples include

```
0xff              // Decimal 255, expressed in hexadecimal
0377              // The same number, expressed in octal (base 8)
0xCAFEBABE        // A magic number used to identify Java class files
```

Each integer type has a corresponding wrapper class: Byte, Short, Integer, and Long. Each of these classes defines MIN_VALUE and MAX_VALUE constants that describe the range of the type. The classes also define useful static methods, such as Byte.parseByte() and Integer.parseInt(), for converting strings to integer values.

Floating-Point Types

Real numbers in Java are represented with the float and double data types. float is a 32-bit, single-precision floating-point value, and double is a 64-bit, double-precision floating-point value. They can represented using dot notation or exponential/scientific notation.

The float and double primitive types have corresponding wrapper classes, named Float and Double. Each of these classes defines the following useful constants: MIN_VALUE, MAX_VALUE, NEGATIVE_INFINITY, POSITIVE_INFINITY, and NaN. To check whether a float or double value is NaN, you must use the Float.isNaN() and Double.isNaN() methods.

Strings

The String type is a class, however, and is not one of the primitive types of the language. A String literal consists of arbitrary text within double quotes. For example:

"Hello, world"

"'This' is a string!"

String literals can contain any of the escape sequences that can appear as char literals. More of sting discussion to be covered later.

Declaring variables and constants

Variables can be initialized at declaration, for example:

```
int taxBracket = 30000;
String message = "Hi!";
```

The difference in case between `int` and `String` indicates that they are different kinds of variables

Sometimes we want to use a constant, a quantity that will not change during the execution of a program

```
final int MAX_STUDENTS = 100;
final int TAX_BRACKET = 30000;
```

The keyword `final` has different meanings when applied to different things, but for these primitive (`int`) values it makes them constant, they cannot be changed.

Note the coding style use of ALL CAPITALS for constants.

Type Conversions

Java allows conversions between integer values and floating-point values. In addition, because every character corresponds to a number in the Unicode encoding, char values can be converted to and from the integer and floating-point types. In fact, boolean is the only primitive type that cannot be converted to or from another primitive type in Java.

There are two basic types of conversions, Widening Conversion and Narrowing Conversion. A widening conversion occurs when a value of one type is converted to a wider type—one that has a larger range of legal values. Java performs widening conversions automatically when, for example, you assign an int literal to a double variable or a char literal to an int variable.

A narrowing conversion occurs when a value is converted to a type that is not wider than it. Doing narrowing conversion is a risky as you can lose data. If you need to perform a narrowing conversion and are confident you can do so without losing data or precision, you can force Java to perform the conversion using a language construct known as a cast. Perform a cast by placing the name of

the desired type in parentheses before the value to be converted. For example:

```
int i = 13;
byte b = (byte) i; // Force the int to be converted to a byte
i = (int) 13.456; // Force this double literal to the int 13
```

when you cast from floating-point values to integer values the fractional part is truncated.

Expressions and Operators

The right hand side expressions is evaluated first and its result is assigned to left hand side variables. The operators are what makes expressions and the type operator used in most cases determines the type of expression.

- Assignment operators: Assignment(=), others are +=, - =, * =, /=, %=
- Arithmetic operators: addition (+), subtraction (-), multiplication (*), division (/), remainder division (%), increment (++), decrement(--),
- Relational or Comparison operators: greater than (>), greater than or equal to (>=), less than (<), less than or equal to (<=), equal to (= =), not equal to (!=)
- logical or boolean operators: NOT(!), AND (&&), OR(||), XOR(^)

chapter 2: INPUT/OUTPUT IN JAVA

2.1 Reading Strings in Java

Initial Specification

Write a program to read a string from the user and echo it on screen. Questions:

- where is the string coming from?
- how will users know that they have to input a string?

Refined Specification

Write a program to accept a string from the keyboard and echo it on screen. A prompt should be used to get the string from the user. The output must go to the screen.

User Interface

```
Input a line of text:
my name is Nahele
Your input was: my name is Nahele
```

Design (Initial)

```
read the string
display the string
```

Design (Refined)

```
read the string
   prompt the user
   get the string
display the string
   display header message
   display string
```

An implementation of this is as follows:

```
// ========================================
//  Written DS 16/03/2011
//  It gets a string from the user and
//  echoes it on screen
// ========================================

import java.io.*;
public class Echo {
```

```
   public static void main (String[] args) throws IOException {
   // must include this, more later

   // create a text input stream
      BufferedReader stdin = new BufferedReader(new InputStreamReader (System.in));
      String message;

      System.out.println("Input a line of text");
      message = stdin.readLine();
      System.out.println("Your input was: " + message);
   }
}
```

In the program above we can see a few new things:

- the line

  ```
  import java.io.*;
  ```

  saying that we intend to use the facilities of the standard package `java.io`, in this case the class `BufferedReader;`

- the statement

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

  which uses the new operator to create a new instance stdin of a BufferedReader class to be able to use one of its methods,`stdin.readLine().`

- the segment

  ```
  class Echo {
     public static void main (String[] args) throws IOException
  ```

  where we see the throwing of an exception. Abnormal behavior or limit circumstances in Java are called exceptions. When reading input it is likely that something will go wrong or a limit is reached, so the method used for reading `stdin.readLine()` 'throws an exception' when this happens. The `readLine()` method throws `IOException`.

- The class Echo can:
  - catch the exception
  - ignore it: in which case it is forced by the compiler to add throws IOException, to inform that it is not intending to handle the exception
- The line

  ```
  String message;
  ```

  declares a String called message to read the user's input value read with the call

  ```
  message = stdin.readLine();
  ```

- The statement

  ```
  System.out.println ("Your input was: " + message);
  ```

prints the output message to the screen using the concatenation operator + to concatenate one string after the other.

2.2 Reading Numbers in Java

Specification

Write a program to add two numbers. Questions:

- where are the numbers coming from?
- are the numbers whole (integers), or they have a decimal part?
- are negatives allowed?
- where should the output go to?

Refined Specification

Write a program to accept two whole numbers, positive or negative, from the keyboard and add them up. The resulting sum will be displayed on the screen.

User interface example

```
Input the first integer number:
5
Input the second integer number:
7
The sum is:  12
```

Design

```
get the two numbers
   get the first number
      show prompt
      read in first number
   get the second number
      show prompt
      read in second number
add them
display the sum
   show prompt
   show sum
```

The following code implements the solution

```
// Written DS 16/03/2011
// This program gets two numbers
// from the user and adds them
```

```
import java.io.*;
public class AddingInts {
   public static void main (String[] args) throws IOException {
      // create a text input stream
      BufferedReader stdin = new BufferedReader
         (new InputStreamReader (System.in));
      String string1, string2;
      int num1, num2, sum;

      System.out.println ("Input an integer number");
      string1 = stdin.readLine();
      num1 = Integer.parseInt (string1);

      System.out.println ("Input another integer number");
      string2 = stdin.readLine();
      num2 = Integer.parseInt (string2);
      sum = num1 + num2;
      System.out.print("The sum is: " + sum);
      System.out.println();
   }
}
```

New concepts have been introduced in this example:

- A Java primitive data type: `int`
- `int` variable declarations: `num1`, `num2`, `sum`;
- Operator addition (`int`): `+`
- The use of the method `Integer.parseInt()` to convert the string read to an `int` variable
- The use of `+` with a `String` and an `int`, where the `int` is converted to a `String`, concatenated and printed:

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

- Use of `System.out.print()` to display `sum` without going to the next line, and of `System.out.println()` to move the cursor to the next line

A couple of things to note in the previous programs:

- As seen above in our two previous examples, reading a string or an int from the keyboard is not simple in Java.
- However, due to the object-oriented nature of Java, it is possible to create a special class (a convenience class) to effect keyboard input and make the job easier. We can create a class `ConsoleReader` to accept different inputs from the keyboard.
- First you need to create a CosoleReader object associated with Standard input (System.in) as in:

  ```
  ConsoleReader console=new ConsoleReader(System.in);
  ```

- Subsequently this object can be used for reading (string, int or double) values directly from the keyboard as in:

  ```
  String line = console.readLine();
  int n = console.readInt();
  double d = console.readDouble();
  ```

- Now we can rewrite the first example above as follows:

For example, we can rewrite the first example above as follows:

```
import java.io.*;
public class Echo {
   public static void main (String[] args)throws IOException {
      // assumes that ConsoleReader.java file below is
      // placed in the same directory

      ConsoleReader console=new ConsoleReader(System.in);
      String message;
      System.out.println("Input a line of text");
      message = ConsoleReader.readLine();
      System.out.println("Your input was: " + message);
   }
}
```

A couple of things to remark here:

- As the ConsoleReader is not a standard class it must be placed in a file named ConsoleReader.java, in the same directory.
- In later chapters we will see how these classes can be placed in a package.

Our second example above can be rewritten as follows:

```
// Written DS 16/03/2011
// This program gets two numbers
// from the user and adds them

import java.io.*;
public class AddingInts {
   public static void main (String[] args) throws IOException {
      // create a ConsoleReader
      ConsoleReader console=new ConsoleReader(System.in);
      // no need to use temporary strings as before
      int num1, num2, sum;

      System.out.println ("Input an integer number");
      num1 = console.readInt();

      System.out.println ("Input another integer number");
      num2 = console.readInt();

      sum = num1 + num2;
      System.out.print("The sum is: " + sum);
      System.out.println();
   }
}
```

In the examples above, there is no need for us to create the text input stream using `BufferedReader`, we don't need to read into a string first and then convert, etc, all that is done *inside* `ConsoleReader` and is hidden from us. In this way classes and objects provide services to others in a safe manner.

Coding Style

- for classes, title case:

```
class ColorsOfTheRainbow ....
class EmployeeManager ...
```

- for variables, start with a lowercase:

```
String string1, string2;
int num1, num2, sum;
```

2.3 A helper class for handling input - ConsoleReader

Brief details of ConsoleReader class

- only one instance variable, a reference to a BufferedReader object
- a constructor which has the same name as the class, used for initializing instance variables when the object is first created
- methods readInt(), readDouble() and readLine() for readingint, double and string values directly The readLine() method handles the possibility of an I/O exception.

```
import java.io.*;
public class ConsoleReader
{
   public ConsoleReader(InputStream inStream) {
      reader = new BufferedReader(new InputStreamReader(inStream));
   }

   private BufferedReader reader;

   public int readInt() {
      String inputString = readLine();
      int n = Integer.parseInt(inputString);
      return n;
   }

   public double readDouble() {
      String inputString = readLine();
      double n = Double.parseDouble(inputString);
      return n;
   }

   public String readLine() {
      String inputLine = "";
      try {
         inputLine = reader.readLine();
      }
      catch(IOException e) {
         System.out.println(e);
         System.exit(1);
```

```
        }
        return inputLine;
    }
}
```

Note: If you dont understand this, for now just dont worry, as we will later cover a full chapter explaining how to create your own classes

Chapter 3: Control Structures

3.1 Stepwise Refinement

Specification: Write a program to manage withdrawals from a bank account. If there are enough funds, the program should accept the withdrawal, otherwise it should reject it.

Questions:

- What is the initial value of the balance?
- What kind of outputs should the program produce?

Refined Specification : Write a program to manage withdrawals from a bank account. The program should first input initial balance and withdrawal amount through keyboard. If there are enough funds, the program should accept the withdrawal, otherwise it should reject it. New balance must be displayed at the end of transaction.

User Interface: there are two possibilities, namely:

```
Enter initial balance: 150
Enter withdrawal: 50
Withdrawal accepted.
New balance: 100
```

```
Enter initial balance: 50
Enter withdrawal: 125
Withdrawal rejected.
New balance: 50
```

Design:

```
get balance
get withdrawal
decide accept or refuse
```

Design (refined):

```
get balance
   display prompt
   get balance value
get withdrawal
   display prompt
   get withdrawal values
decide accept or refuse
   if balance greater than or equal to withdrawal, accept
      display accept message
      calculate new balance
      display new balance message and value
   otherwise reject
      display reject message
      display new balance message and value
```

Implementation:

```
// Written DS 19/03/11
// accepts an initial balance from the keyboard
// and a withdrawal; it accepts or rejects
// the operation according to the values

import java.io.*;
class AccountManagement {
   public static void main (String[] args) throws IOException {
      // create a text input stream, the standard way
      BufferedReader stdin = new BufferedReader (new InputStreamReader
(System.in));
      String initString, withdrawString;
      int initialBalance, withdrawal;

      System.out.println ("Enter initial balance");
      initString = stdin.readLine();
      initialBalance = Integer.parseInt (initString);
      System.out.println ("Enter withdrawal");
      withdrawString = stdin.readLine();
      withdrawal = Integer.parseInt (withdrawString);

      if (initialBalance >= withdrawal)  {
         System.out.println("Withdrawal accepted");
         System.out.print("New balance: ");
         System.out.println(initialBalance - withdrawal);
      }
      else  {
         System.out.println("Withdrawal rejected");
         System.out.print("New balance: ");
         System.out.println(initialBalance);
      }
   }
}
```

We introduced here:

- the `if ... else` statement
- the "greater than or equal to" operator `>=`
- The use of curly braces to make a *block*

```
if (initialBalance >= withdrawal)  {
   System.out.println("Withdrawal accepted");
   System.out.print("New balance: ");
   System.out.println(initialBalance - withdrawal);
}
```

so several statements are considered as one.

Any collection of statements between braces is called a block, which could be the *empty block* `{}`.

There are uses for the empty block in Java, such as when we want to trap an exception but do nothing with it.

> **Note:** The use of proper indentation to indicate statements that are dependent on others. The compiler ignores the indentation, but the code is easier to read by humans.

3.2 The if ... else statement

The `if...else` statement is the main statement for decision making within a program, such as the one above.Its general form is:

```
if (expression)
   statement            // executed if condition is true
else
   statement            // executed if condition is false
```

or, if using blocks,

```
if (expression) {
   statement
        .......         // executed if condition is true
   statement
}
else {
   statement
        .......         // executed if condition is false
   statement
}
```

The Rules Are

- *expression* must be a `boolean` expression, that is, an expression that evaluates to `true` or `false`. If *expression* is `true` the first statement is executed; otherwise the second statement is executed. Statements can be more than one if they are included in a block.
- The individual `statements` to be executed are statements, so they include a `';'` at the end;
- the `else` is optional. If not present, the syntax is:

```
if (expression)  {
   statement
   .......      // executed if condition is true
   statement
}
```

In this case, the statements are executed if *expression* is `true`, and completely skipped if it is `false`. For example, a code segment to detect if a *positive* `int` being entered is a digit would be:

```
if (number < 10)
   System.out.println("The number entered is a digit");
```

> Note: The difference between the simple `if` and the `if...else` is that in the first case there is no alternative action; `if` the condition is `false`, the execution continues immediately after the if statement, no action is taken.

3.3 Nested if ..... else Statements

Since `if` is a Java statement, we can have an `if` within another `if`. This approach is used to implement decisions within decisions:

```
if (Mark >= 40)         //  pass the subject
   if (Mark >= 70)
      System.out.println("Well done!");
   else
      System.out.println("Passed the subject");
```

or

```
if (Mark >= 40)         //  pass the subject
   if (Mark >= 70)
      System.out.println("Well done!");
   else
      System.out.println("Passed the subject");
else                    //  not passed
   if (Mark>=35)
      System.out.println("You'll get another chance!");
   else
      System.out.println("I am sorry, but you fail");
```

3.4 Multway Decisions

The previous example shows a situation where there are more than two options to choose from. This is called a *multiway* decision. Another way of writing the same code could be

```
if (Mark >= 70)
   System.out.println("Well done!");
else if (Mark >= 40)
   System.out.println("Passed the subject");
else if (Mark>= 45)
   System.out.println("You'll get another chance!");
else
   System.out.println("I am sorry, but you fail");
```

Both code segments produce the same results, but the second form is easier to read, and it is usually preferred by programmers.

Summarising Decision Making so far

```
if (Boolean_expression)
        statement;      // simple decision
```

```
if (Boolean_expression)
  statement;
else
   statement;           // else is optional
```

```
if (Boolean_expression1)
   statement;
else if (Boolean_expression2)
   statement;           // 0 or more else if
else
   statement;           // else is optional
```

3.5 The Conditional Operator: ?

The conditional operator is used as a shorthand for an `if...else` statement. Let's consider the following code to calculate the minimum of two numbers `x` and `y`:

```
if (x < y)
   minVal = x
else
   minVal = y;
```

Using the conditional operator this can be written:

```
minVal = (x < y) ? x : y;
```

The general case is:

condition ? *YesExpression* : *noExpression*;

The condition is evaluated first; if true the value of the entire expression is yesExpression, otherwise the value is *noExpression*. In this way, we can assign the value of the expression to a variable as we did above. By the way, the brackets above are not really necessary since the precedence of the ?: operator is above that of the assignment (on purpose, of course), so they are used only for clarity. The operator is very useful when you have to choose between two simple alternatives; it can even be included in expressions,

```
System.out.println("The number is: " + ((n % 2) == 0 ? "even" : "odd"));
```

Question: which of the above brackets are necessary?

3.6 The switch Statement

The following example illustrates the use of the Java `switch` statement for multiway decisions, as an alternative to nested `if...else`. The following code prints the month name corresponding to a month number:

```
switch (month)  {
   case 1 : System.out.println("January");
      break;
   case 2 : System.out.println("February");
      break;
   case 3 : System.out.println("March");
      break;
   case 4 : System.out.println("April");
      break;
   case 5 : System.out.println("May");
      break;
```

```
   case 6 : System.out.println("June");
      break;
   case 7 : System.out.println("July");
      break;
   case 8 : System.out.println("August");
      break;
   case 9 : System.out.println("September");
      break;
   case 10 : System.out.println("October");
      break;
   case 11 : System.out.println("November");
      break;
   case 12 : System.out.println("December");
      break;
   default : System.out.println("Not a valid month");
      break;
}
```

Output

```
Enter a month(1-12): 5

May
```

The rules for the `switch` Statement are:

- The selector must be an integer-type expression
- If the selector matches any of the values in the cases, the execution continues there. The `break` is needed to avoid the execution of the statements following the match. When the execution finds a `break` the entire `switch` is terminated and execution continues with the first statement after the switch. (More on `break` later).
- If there is no match on any of the cases, control transfers to a `default` case. If there is no `default` case the entire `switch` is terminated and execution continues with the first statement after the `switch`.

Note: a common danger is to forget the `break` after a `case`.

Let's consider another example:

```
switch (month)  {
   case 1 :
   case 2 :
   case 3 : System.out.println("First Quarter");
      break;
   case 4 :
   case 5 :
   case 6 : System.out.println("Second Quarter");
      break;
   case 7 :
   case 8 :
   case 9 : System.out.println("Third Quarter");
      break;
   case 10 :
   case 11 :
   case 12 : System.out.println("Fourth Quarter");
```

```
      break;
   default : System.out.println("Wrong Input!");
      break;
}
```

Note how the `break` statements have been placed so several alternatives are handled by the same code. The execution 'falls through' to the code handling it.

```
switch (expression)   {
   case constant1 : statement;
      break;
   case constant2 : statement;
      break;
   case constant3 : statement;
      break;
      ..................
   default : statement;
      break;
}
```

3.7 Definite Iteration

Repetition (*iteration*) is to used make a program repeat a set of actions a number of times. There are two types of iteration:

Definite: the number of times is known in advance, either when writing the program or when the program is running.

Indefinite: the number of times cannot be determined from the specification or when the program is running.

Definite iteration

As an example, let's consider the problem of averaging three `int` numbers. To do that we need first to add the three numbers up and then divide the sum by 3. If the numbers are being entered by the user via the keyboard, the following `for` loop does the job:

```
// assume appropriate declarations of
// i, num, inString, sum, etc.

sum = 0;
for ( i = 1 ; i <= 3 ; i++)  {
   System.out.println ("Input an integer number");
   inString = stdin.readLine();
   num = Integer.parseInt (inString);
   sum += num;   // the same as sum = sum + num
}
```

At the end of the `for` loop each number has been entered into `num` and added to `sum` to obtain the total. After that it is still necessary to divide by 3. Let's write a complete Java program to do this.

Specification: Write a program to calculate the average mark of a student for three assignments, each one marked out of 10.

Refined Specification: Write a program to accept three `int` values between 0 and 10 for a student's three assignments, and calculate the student's average mark for the assignments. Display the average value to the screen. NB: the average value is not necessarily an `int`.

User Interface (Example)

```
CALCULATION OF ASSIGNMENTS AVERAGE
Enter mark of assignment 1:  5
Enter mark of assignment 2:  8
Enter mark of assignment 3: 10

The average mark is: 7.66
```

Design:

```
calculate the sum of the marks
process the average
```

Design (revisited):

```
calculate the sum of the marks
   for all assignments
      prompt for mark
      get mark
      add to previous total
process the average
   calculate the average
   display the average
```

```
// Written DS 20/03/11
// This program gets three assignment marks
// for a student, adds them up
// and calculates their average
import java.io.*;
public class AverageMarks {
   public static void main (String[] args) throws IOException {
      // create a text input stream
      BufferedReader stdin = new BufferedReader (new InputStreamReader
(System.in));
      String inString;
      int i, num, sum = 0;
      for ( i = 1 ; i <= 3 ; i++) {
         System.out.println ("Enter mark of assignment " + i + ": ");
         inString = stdin.readLine();
         num = Integer.parseInt (inString);
         sum += num;  // accumulate total
      }
      //print the average
      System.out.println("The average is: " + (double)sum / 3);
   }
}
```

Note that the value of `sum` has been *cast* (type converted) to a `double` to make sure that we don't get the integer result but the floating point one.

Some things to note here:

- The general form of a Java `for` loop is:

```
for (initial; test ; increment)
   statement
```

- `initial`, `test` and `increment` are the expressions that control the loop. Any of them, or all of them, are optional and can be omitted. For example, an infinite loop can be written:

```
for(  ;  ;  )
   statement
```

  We'll see more cases like this later on.
- *`Statement`* can be any Java statement, including a block statement, `if...else`, `for` loop, etc.
- it is possible, and very convenient, to declare the loop variable in the `for` statement itself:

```
for ( int i = 1 ; i <= 3 ; i++)
```

  In this case, the lifetime of variable `i` ends when the loop finishes; that is, any mention of `i` outside the loop will fail to compile.
- The behaviour of the `for` loop is as follows:

1. the initial expression is evaluated
2. the `test` expression is evaluated. If false, the statement is not executed and control passes to the first statement after the loop. If `test` is true the statement is executed and then the `increment` is performed. This step is repeated as long as the `test` expression remains true

> Note: often the values for the expressions must be determined when the program runs, by getting them from the user, for example. It is still definite iteration.

The for loop can be done 'in reverse', by initialising at the highest end of the loop and decrementing. For example:

```
for ( int i = 3 ; i >= 1 ; i--)
```

performs the loop from 3 down to 1. Let's consider the problem of writing a conversion table from cubic inches to cubic centimeters, down from 20 cubic inches to 2 cubic inches, every 2 inches. The main code segment could be written:

```
double cubicCm; // cubic centimetres
int finish, step;
// constant conversion factor from cubic in to cubic cm
final double CONV_FACTOR = 1000.0/61.0;
finish = 10; step = 2;

System.out.println("CONVERSION TABLE FROM CUBIC INCHES" + " TO CUBIC CM");
for ( int i = finish; i > 0 ; i--) {
   cubicCm = (i * step) * CONV_FACTOR;
   System.out.println(cubicCm);
}
```

The output of the code is:

```
CONVERSION TABLE FROM CUBIC INCHES TO CUBIC CM
327.86885245901635
295.08196721311475
262.2950819672131
229.50819672131146
196.72131147540983
163.93442622950818
131.14754098360655
98.36065573770492
65.57377049180327
32.78688524590164
```

Say we want to write code to produce the following output:

```
 1
 2  3
 3  4  5
 4  5  6  7
 5  6  7  8  9
 6  7  8  9 10 11
 7  8  9 10 11 12 13
 8  9 10 11 12 13 14 15
 9 10 11 12 13 14 15 16 17
 10 11 12 13 14 15 16 17 18 19
```

Given that the `for` loop is a statement like any other, it is possible to nest a `for` loop within another `for` loop:

```
for (int i = 1; i <=10; i++) {
   for (int j = 1; j <= i; j++)
      System.out.print(i + j-1);
   System.out.println();
}
```

Note that the inner `for` loop affects only a single statement, and that braces are necessary for the outer `for` loop to get the correct output.

3.7 Indefinite Iteration

Often the number of times to iterate can't be determined a priori, but depends on a certain condition becoming `true` or `false`. This kind of iteration can be handled by the `while` loop:

```
// 0 or more iterations
while (boolean_expression)
   statement;
```

For example:

```
while (Answer != 'Q')
   total++;
```

A classical case is to handle an unknown number of items input by the user. For example, to do some processing while there are still items such as students, employees, etc., to process.

Specification: write a program to accept characters as input, and quit when a `'Q'` is typed, printing how many characters have been typed.

Refined Specification: write a program to accept characters as input, and quit when a `'Q'` or a `'q'` is typed, printing how many characters other than the `'Q'` or `'q'` have been typed.

User interface:

```
This program counts the number of characters entered.

Input a character, 'Q' or 'q' to quit:
r
Input a character, 'Q' or 'q' to quit:
e
Input a character, 'Q' or 'q' to quit:
s
Input a character, 'Q' or 'q' to quit:
P
Input a character, 'Q' or 'q' to quit:
J
Input a character, 'Q' or 'q' to quit:
q
The number of characters entered is:    5
```

Design:

```
initialisation
while input not a q or a Q
   process character
display total
```

Design revisited:

```
initialisation
   show initial prompts
while input not a q or a Q
   process character
      prompt for a character
      get character
      add 1 to total
display total
```

Code:

```
// DS 20/03/2011
// Illustrates indefinite iteration
// by accepting characters until a 'q' or 'Q' is typed

import java.io.*;
public class CharTest {
   public static void main (String[] args) throws IOException {
      // create a text input stream and initialise
      BufferedReader charInput = new BufferedReader (new InputStreamReader
(System.in));
      char answer;
      int charTotal = 0;
```

```
      System.out.println("This program counts the " + "number of characters
entered.");
      System.out.println("Input a character, " + "'Q' or 'q' to quit:");
      answer = charInput.readLine().charAt(0);  //look at this

      while (answer != 'Q' && answer != 'q')  {
         charTotal++;
         System.out.println("Input a character," + " 'Q' or 'q' to quit:");
         answer = charInput.readLine().charAt(0);  //look at this
      }
      System.out.println("The number of characters" + " entered is:");
      System.out.println(charTotal);
   }
}
```

Notice the trick of prompting and getting the first answer before the `while` loop and then at the end of it to get the behavior we want. Also, the fact that we use a `BufferedReader`method (`readLine()`), even though we only read one `char`, to deal properly with the end of line.

The behavior of the `while` loop is as follows:

- the condition in the `while` is evaluated first
- if `false`, the execution continues in the first statement after the loop (the loop is skipped)
- if `true`, the body of the loop, i.e. the (possibly a block) statement following the `while`, is executed
- if the body was executed, the condition is evaluated again, if `true` the body is executed again, the condition re-tested, and so on.

Note that if the first time the condition is tested the result is `false` the `while` loop is not executed at all, that is, the statement in the `while` is skipped completely. The `while` loop is termed '0 or more repetitions' because it could happen that the loop is not performed at all, that is, performed 0 times.

It is possible to use a loop to implement 1 or more iterations, avoiding the double prompting. The code above could be re-written:

```
do {
   System.out.println("Input a character, " + "'Q' or 'q' to quit:");
   answer = charInput.readLine().charAt(0);
   if (answer == 'Q' || answer == 'q') break;
      charTotal++;
} while (true);
```

The execution continues until the user inputs a `'Q'` or a `'q'`, the `break` statement returns control immediately after the loop. Note that it is a potentially 'infinite' loop, it will only exit when the user inputs a quit character.

The general form of a `do` loop is:

```
do {
   statements;
} while (condition);
```

The difference between both loops is that the `while` loop will be skipped altogether if the condition is `false` the first time the loop is executed. The `do` loop is executed at least once, so it is called a '1 or more repetitions' loop.

Perhaps a more standard example of the `do` loop is to control and validate users' input, as in:

```
do {//  menu to accept only a, b or q
    //  Note: it doesn't handle capitals properly!
   System.out.println("Choose an option:");
   System.out.println("a) Add an employee");
   System.out.println("b) Display an employee");
   System.out.println("q) Quit");
   answer = charInput.readLine().charAt(0);
}  while (answer != 'a' && answer != 'b' && answer != 'q');
```

The only way to exit this loop is to enter a legal option: `'a'`, `'b'`, or `'q'`. At the exit of the loop we know that the input is the one we want.

Summary of indefinite iteration

- Multiple exit points are possible but ill advised! Sometimes it becomes difficult to find out when the loop is exited, especially if the loop is long.
- The iterations above are indefinite repetition/iteration statements; the number of iterations is not specified in advance.
- The loop terminates when the boolean expression changes.
- Only statements in the loop can terminate the loop.

Beware of infinite loops! Make sure that the expression will eventually change to terminate the loop.

```
while (boolean_expression)
   statement    // 0 or more iterations
```

```
do  {           // repeat loop
   statement
}  while (boolean_expression);
```

3.8 The `break` and `continue` Statements

We have used already the `break` statement to break out of a `switch` and a `while` or `do...while` statements. It can be used to break out of any looping statement, that is, the ones mentioned above and the `for` loop, and return control to the first statement after the loop.

There is also a `continue` statement that can be used inside a loop. It is less dramatic than the `break` in the sense that it doesn't cause the loop to terminate but it causes the flow of control to pass to the next iteration of the loop.

For example, say we want to do a process where employees whose salary is less than $20,000 are given a 10% bonus.

```
//  assume proper declarations and initialisation
//  employee numbers and salaries are entered
```

```
//  on different lines

int empNumber = 0;  // value 0 forces entry into the while
double empSalary = 0;

while (empNumber != -1)  {
   System.out.println("Input next employee number," +
                                        " -1 to quit");
   empNumber = Integer.parseInt(stdin.readLine());
   if (empNumber != -1) {
      System.out.println("Input next employee salary");
      empSalary = Integer.parseInt(stdin.readLine());
   }

   if (empSalary >= 200000)
      continue;  // jump to next iteration

   empSalary *= 1.1;  // 10% bonus
   System.out.println("The new salary is: " + empSalary);
}
```

This (somewhat contrived) example shows that when an employee's salary is more than Tsh. 200,000, the loop continues with the next iteration. It also shows:

- the initial value of 0 for `empNumber` to force the entry into the loop
- the use of -1 as a *sentinel*, a special value different from any normal value that serves as a flag to end the input.

3.9 Worked Examples

Example 1:

```
// DS, 20/03/11
// Counts the number of days with no rain, the day
// with highest rainfall, the total rain and the average
// rain in a given period
import java.io.*;

public class RainManager {
   public static void main (String[] args) throws IOException {
      // create a text input stream
      BufferedReader stdin = new BufferedReader (new InputStreamReader
(System.in));
      String inString;
      int rainTotal = 0;
      int rainToday ;
      int rainHighest = 0;
      int dayHighest = 0;
      int zeroCount = 0;
      int numOfDays;
      System.out.println("Enter number of days to process");
      inString = stdin.readLine();
      numOfDays = Integer.parseInt (inString);

      System.out.println("Enter rainfall for each day");
```

```
      for (int i = 1 ; i <= numOfDays ; i++)  {
         System.out.println("Day " + i + ": ");
         inString = stdin.readLine();
         rainToday = Integer.parseInt (inString);
         rainTotal += rainToday;  // accumulate total

         if (rainToday > rainHighest)  {// new highest rain
            rainHighest = rainToday;
            dayHighest = i;
         }

         if (rainToday == 0)
            zeroCount++;

      }  //  for loop

      //  print the average
      System.out.println("Average rainfall for the period " + " was
                         " + (double) rainTotal / numOfDays + " mm");
      System.out.println("Highest daily rainfall was " + rainHighest + " mm");
      System.out.println("Highest rainfall occurred on day " + dayHighest);
      System.out.println("There was zero rainfall on " + zeroCount + " days");
   }  //  main
}
```

Example 2:

```
// DS, 20/03/11
// Reads Price and Amount paid until Price <= 0.
// Computes the U.S. Change in Quarters (25c),
// Dimes (10c) , Nickels (5c) , and cents.
// Amount of Change must not exceed 99 cents.

import java.io.*;

class ChangeMoney  {
   public static void main (String[] args) throws IOException {
      // create a text input stream
      BufferedReader stdin = new BufferedReader (new InputStreamReader
(System.in));
      final int QUARTER = 25;
      final int DIME = 10;
      final int NICKEL = 5;
      int price, amount, change;
      int numOfQuarters = 0, numOfDimes = 0;
      int  numOfNickels = 0, numOfCents = 0;

      System.out.println("Please input price in cents," + " 0 to quit:");
      price = Integer.parseInt (stdin.readLine());


      while (price > 0)  {
         System.out.println("Please input amount paid" + " in cents :");
         amount = Integer.parseInt (stdin.readLine());

         change = amount - price;
```

```
         if (change > 99 || change < 0)
            System.out.println("Incorrect data, change" + "must be in [0,99]
cents");
         else if (change == 0)
            System.out.println("No Change");
         else {
            // compute Quarters, Dimes, Nickles, cents
            numOfQuarters = 0;
            while (change >= QUARTER) { // still QUARTERs?
               numOfQuarters++;
               change -= QUARTER;   // take QUARTER away
            }
            numOfDimes = 0;
            while (change >= DIME) { // still DIMEs?
               numOfDimes++;
               change -= DIME;
            }
            numOfNickels = 0;
            while (change >= NICKEL) { // still NICKELs?
               numOfNickels++;
               change -= NICKEL;
            }

            numOfCents = change;  // leftover are cents

            System.out.println("Change given as " + numOfQuarters +  " quarters " +
numOfDimes +
                                                  " dimes " + numOfNickels + "
nickles and
                                                  " + numOfCents + " cents");

            System.out.println("Please input price in cents, " + "0 to quit:");
            price = Integer.parseInt (stdin.readLine());
         }
      } //  while
   }
}
```