Department of Computer Science
New Mexico State University

CS 272 – Fall 2004

# A Quick and Dirty Overview of Java and Java Programming

*Enrico Pontelli and Karen Villaverde*

# Introduction

## Objectives

In this document we will provide a very brief overview of the main concepts of the Java programming language, along with a number of examples.

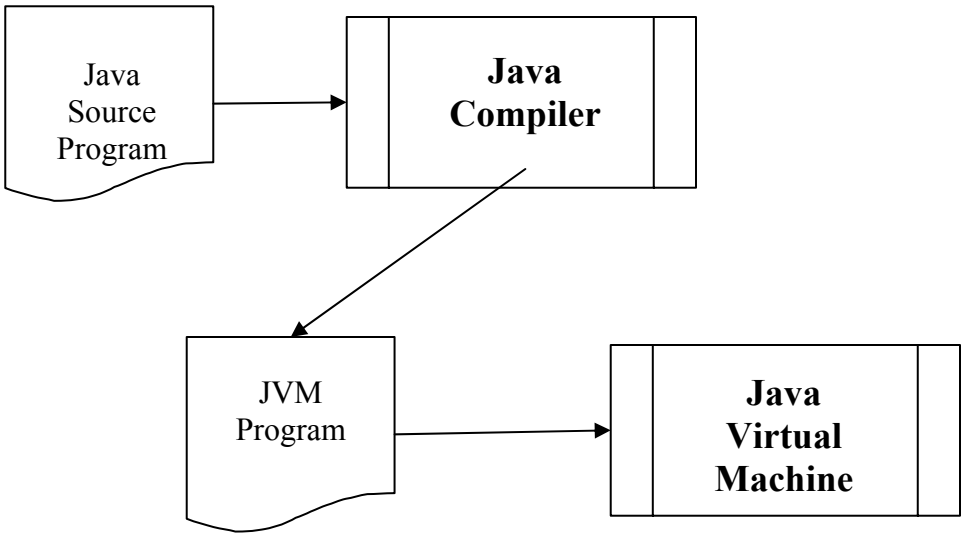
The purpose of the document is not to teach Java, but to allow someone who has already learned Java before to review some of the basic concepts. The students are encouraged to type the various sample programs and ensure that they work properly.

## Compiling and Running Java Programs

Before we start with our overview of the basic concepts of Java, let us recall the basic mechanisms that are employed to execute Java programs.

The assumption is that you are currently using a standard SDK installation of Java in a Linux-type environment. If you are using a different environment (e.g., under Windows), some of this might not apply.

Java is a *compiled language*. This means that, in order to execute a program, you will first need to go through a *translation* process (*compilation*). The compilation process is aimed at translating the Java source program into another equivalent program written in a lower level language. In the case of Java, the lower level code is expressed in a language called *Java Virtual Machine code* (JVM code). This code is then executed by a piece of software, called the Java Virtual Machine (JVM). The JVM executes one instruction at a time from its input program. This process is illustrated in the figure below.




These two steps (compilation and execution) have to be explicitly requested by issuing appropriate commands in a terminal window.

The compilation is requested by using the command **javac** which requests the execution of the java compiler. The command requests one input, which is the name of a file that contains a Java source program. By default, we expect these files to be named with an extension '.java' (e.g., FirstProgram.java). The typical compilation command line is:

**javac** FirstProgram.java

If there are some syntax errors in your program, the compiler will return a list of them, and you will have to fix them and recompile the program. If there are no errors, the compiler will create a new file which contains the result of the translation (a program expressed in JVM code). The resulting file typically has the same name with the exception of the extensions, which is '.class' (e.g., FirstProgram.class).

At this point you are ready to execute the program. To accomplish that you will need to start the Java Virtual Machine and feed the JVM code as input to it. The Java Virtual Machine is just another program, that can be executed by specifying its name. The name of the JVM is simply **java**. Thus, if we request the execution of the command **java** and we provide as parameter the name of a file that contain JVM code, we will accomplish the goal of executing such JVM code. In our example, the command line to be used is:

**java** FirstProgram

As you can see, when we provide the input to the JVM, we do not need to specify the '.class' extension in the file name (this is automatically added by the JVM when it accesses the file).

# Basic Types, Expressions, and Control Structures

## A Sample Java Program

Let us start by showing a sample Java program, that we will then modify to illustrate the various concepts presented.

```
// A very silly program that does almost nothing

class FirstProgram
{
    // this is the main part of the program
    public static void main(String args[])
    {
        int x,y,z;

        x = 6;
        y = 2;
        z = x+y;
        System.out.println("The result of ",x"+"+y" is "+z);
    } // end of main
} // end of class
```

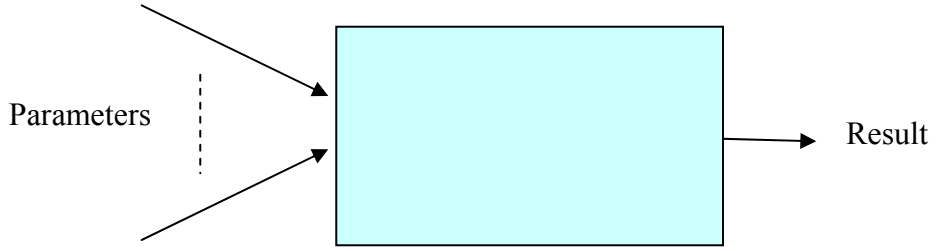
Each program we will write will be composed of *classes*. We will extensively talk about classes later in the course, but for the moment you can think of a class as a container which encloses all the various functions and data that you will use in your program. In our example, the container (called FirstProgram) contains only one method – the **main** method. Every program has to contain exactly one method called **main**. This method will be automatically executed when you start your program – thus, the **main** method represents the point where the execution of your program is supposed to start.

Every method is structured in two parts. The first part, called the *Header* of the method, is a single line that essentially describes how the method is supposed to be used. The format of the Header is:

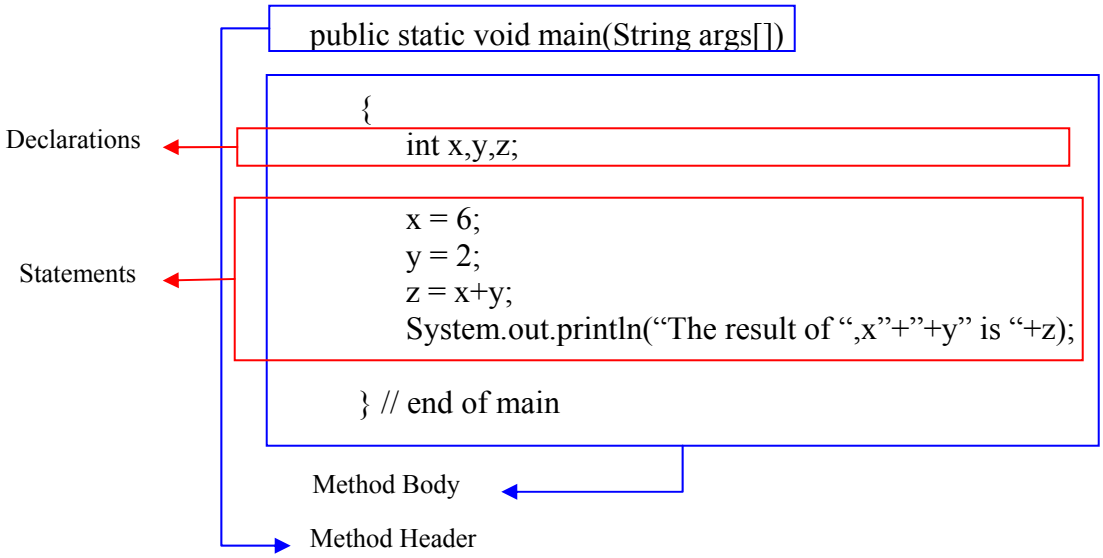
ReturnType Name of Method ( Comma-separated List of Parameters )

The Name of Method indicates what name we are giving to the method; this name will be used to identify the method when we request its execution. The only exception is the method called **main**: we never explicitly request its execution, its execution starts automatically when we initiate executing the program.

The Parameters represent the inputs that we wish to provide to the method; the Return Type describes what type of result the method will produce upon execution. For simplicity, you can think of a method as a black box, that takes a number of inputs (the parameters) and produces one result (and Return Type is telling us what is the type of the result).




The second part of a method is called the *Body* of the method; the body is enclosed between braces ('{' and '}') and it is itself organized in two parts. The first part is a collection of *declarations*, typically used to describe what variables are going to be employed in the method, and a collection of *statements*. The statements are the actual instructions that describe what steps should be taken to perform the action that the method is supposed to perform. If we take the above example, we obtain the situation illustrated in the following figure.




## Variables

All the data used by a program have to be available in memory in order to be used. The data are stored in containers called *variables*. Each variable can store one entity (we will call them *objects*, although this term will be refined in the successive sections). Each variable, in order to exist, has to be *Declared* in the declaration section of a method. We can have also variables declared outside of the methods, but we will deal with these later on.

Every variable in the program has five attributes that describe its properties:

1. name: this is the name that is used to refer to the variable;
2. type: this information describes what kind of values we will be allowed to place in the variable and what type of operations we will be able to perform;
3. value: this is the current content of the variable; at any point in time, the variable has exactly one value;
4. scope: this is the part of the program where a variable can be used. In Java we have two types of scopes:
    a. *Local:* the variable is declared inside a method; in this case the variable can be used exclusive within that method;
    b. *Global:* the variable is declared inside the class but outside of any specific method; in this case, each method in the class can make use of such variable.

5. lifetime: this is the period of time between the creation of the variable and its destruction. We typically distinguish between *Static* variables – created when the program is started and destroyed only when the whole program is finished – and *Dynamic* variables – created during the execution of the program, and possibly destroyed before the end of the program.

Let's see some examples:

- A typical declaration of a variable is

  ```
  int x;
  ```

  which declares a variable named x and of type int (i.e., it will store an integer number). By default, the initial value of the variable is zero. If we want to have a different initial value we can explicitly indicate it:

  ```
  int x = 6;
  ```

- if the variable is declared inside a method, e.g.,

  ```
  int mymethod ( )
  {
      int x = 6;
      ....
  }
  ```

  then the variable has a scope limited to the method called mymethod – it can be used only inside this method and nowhere else. If instead we declare the variable outside of the method, e.g.,

  ```
  class FirstProgram
  {
      int x = 6;
      ...
  }
  ```

  then the scope of the variable x is the entire class called FirstProgram; this means that any method inside that class will be able to access and use the variable x.

- A variable that is local to a method, has a lifetime that corresponds to the lifetime of the method. This means that the variable is created exclusively when the method is

started, and it will be destroyed once the method is finished. Consider the following simple example:

```
class FirstProgram
{
    public static void main (String args[ ])
    {
        ....
        PrintResult();
        ....
    }

    public static int PrintResult()
    {
        int x = 6;
        ...
    }
}
```

In this case, the variable x belongs to the method PrintResult; it will be created only when the method is called (e.g., when the call to PrintResult is performed inside the main) and it will be destroyed once the method PrintResult is left.

- On the other hand, a variable that is declared within the class will be immediately created when the class is activated (e.g., when we start executing the program, if the class contains the main method).

## Primitive Types

Each data item in the program has a type. The type defines what are the possible values that the item can assume as well as it indicates what operations we can apply to such item. Thus, a type is always composed of these two components:

- a Domain: the set of values that belong to the type
- a collection of operations that we can apply to objects of the type.

In this section we will focus on a class of types called *Primitive Types*. These are built-in data types that the language. They are characterized by the fact that the domains of these data types contain *Atomic* values – i.e., the values are treated as atomic entities (and they are never "decomposed" into components).

These can be classified into four groups: Boolean types, character types, integer types, and floating point types.

## Boolean Types

Java provides only a single Boolean type, called **boolean**. The domain of this data type contains only two values, denoted **true** and **false**. The operations allowed on objects of this type correspond to the traditional operations on Boolean values, e.g., logical and, logical or, negation, etc. These are denoted by the symbols && (logical and), || (logical or), and ! (negation). For example, the Boolean formula $a \wedge (b \vee \neg c)$ is represented in Java as

```
a && ( b || (! c))
```

## Character Types

Java provides a single type to represent characters, called **char**. The domain of this data type is the set of all characters belonging to the Unicode standard. In particular, the ASCII character set is a strict subset of the Unicode standard. Character constants can be represented in two different ways:

- using the symbol representing the character, enclosed between single quotes; e.g., the character a is represented as 'a'
- using the numeric code associated to the character by the Unicode standard. For example, the character 'a' is represented by the numeric code 97.

The two notations are interchangeable. For example, the following assignments are both valid and accomplish the same effect:

```
char x = 'a';

char x = 97;
```

For the moment, the only immediate operations on characters that we will consider are the standard operations on arithmetic types (increments and decrements). For example, if we want to print all the alphabetic characters we can write:

```
a=97;

for (int i = 0; i<26; i++)

    System.out.println("Character: "+(char)(a+i));
```

## Integer Types

Integer types have a domain which contains a subset of the set of integer numbers. Java provides different types to represent integers, and their differences are exclusively in the range of integers that are offered. The types are:

- **byte**: it is stored in 8 bits and has a domain including all the integers from -128 to +127
- **short**: it is stored in 16 bits and represents integers in the range -32,768 to +32,767
- **int**: it is stored in 32 bits and represents the integers in the range -2,147,483,648 to +2,147,483,647
- **long**: it is stored in 64 bits and represents integers in the range -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807

The operations allowed are the traditional arithmetic operations (addition, multiplication, etc.). It is important to observe that by default all the operations between integers return integers as result. Thus, the expression 5/2 returns the value 2.

### Floating Point Types

Java offers a number of data types that allow to store non-integer numbers. The types differ in the amount of memory employed – leading to a different level of precision (i.e., number of digits that can be stored). Java offers the following types:

- **float**: 32 bits
- **double**: 64 bits

The operations allowed are the usual arithmetic operations. Observe that these are operations between floating point numbers, that produce floating point numbers as result. In this case, the result of the expression 5.0 / 2.0 is 2.5.

## Expressions

The simplest type of expressions that one can build are expressions composed of a single value (constants). A numeric constant is expressed using the traditional notation. Integer numbers should be written without leading zeros (a number with a leading zero is automatically assumed to in octal form, while a number that starts with 0x is assumed to be in hexadecimal format). Floating point constants can be expressed either in the traditional floating point format, e.g., 12.352, or using scientific notation, e.g., 1.232e+3 (which means $1.232x10^3$).

Character constants are expressed either using the symbolic notation and the single quotes, e.g., ‘X’ or ‘b’, or using the numeric notation based on Unicode. Special symbols are used to represent certain unprintable characters, such as the newline character (‘\n’), the tabulation character (‘\t’), the single quote character (‘\’’).

More complex expressions are built using constants, variables, and operators. The simplest type of operator employed is the assignment operator, which allows to store the result of an expression in a variable. The syntax of an assignment is

variable name = expression

The expression is expected to produce a value of the same type as the variable. If this is not the case, then a type conversion process is applied, to change the value of the expression so that it has the same type as the variable. Observe that an assignment statement is itself an expression, which returns as result the same value that is stored in the variable. This allows us to write things like:

```
b = (a = 6) + 2;
```

which stores 6 in a and 8 in b.

More complex expressions can be obtained by using operations allowed by the types of the data used in the expression; for example,

```
a + (b*3-c)/d
```

The order of execution of the operations is determined by the following rules:

- parenthesis: expressions within parenthesis are evaluated first
- precedence: in absence of parenthesis, operators with highest precedence are applied first. For example

  ```
  c+e*d
  ```

  the produce e*d is done first, followed by the addition.
- associativity: when we have multiple operations with the same precedence level, the associativity rules determine in what order they should be executed. Most operations (e.g., +, *, -, /) are left-associative, which means that they are executed from left to right. There are some operators that are right-associative, i.e., they are executed from right to left. A typical example of a right-associative operator is the assignment operator =; thus, the expression

  ```
  a = b = c = 6
  ```

  is evaluated from right to left:

  ```
  a = ( b = ( c = 6) )
  ```

## Relational Operators

Java provides a number of relational operators, i.e., operators that are used to perform comparison between results of expressions. These operators compare two values and produce a value of type Boolean as result. The traditional relational operators are:

- == used to compare for equality
- != used to compare for not-equality

- > used to test for greater-than
- < used to test for less-than
- >= used to compare for greater-or-equal-than
- <= used to compare for less-or-equal-than

### Type Conversions

Whenever expressions with different types are combined within a larger expression, we are faced with the problem of deciding which operation to perform. The operations provided by Java require the arguments to have the same type. E.g., the operation + wants either both arguments of type byte, or both of type short, or both of type int, etc. When expressions of different type are used, an automatic type conversion is applied (commonly called *coercion*). The coercion rules is to always convert from a lower type towards a higher type, according to the following scheme:

- byte (lowest)
- short
- int
- long
- float
- double (highest)

The fundamental rule is to perform the conversion in such a way to reduce the risk of loosing data (e.g., if we perform a conversion from float to int, then we run the risk of loosing the decimal part of the number).

It is important to remember that the conversion is automatically applied when operands of different types are present around an operation. Thus, if we write

```
5 / 2.0
```

the operation has operands of different type (5 is a byte and 2.0 is a float); the 5 is coerced to a float object (5.0) and then the operation between two floats is performed, leading to the result 2.5.

The programmer is also allowed to request explicit type conversions, which can work in any direction (violating also the rule used in the case of coercion). A type conversion is requested by placing the name of the type in front of the value that we want to convert. For example,

```
(int)5.0 / 2
```

leads to the result 2 (since 5.0 is converted to the integer 5, and then an operation between two integers is applied, which produces again an integer result).

## Statements

The body of a method contains a collection of statements, used to describe the steps required to implement the functionality of the method. Java, as most programming languages, provides different types of statements. We classify statements into two main classes:

1. elementary statements
2. complex statements

Complex statements are built by assembling together other statements and executing them according to a predefined ordering.

### Elementary Statements

In Java we have two forms of elementary statements:

- assignments
- method invocations

We have already seen examples of assignment statements. A method invocation requests the execution of a method. The typical format (for the moment…) of a method call is

method name ( parameter values )

which requests the execution of the method called method name, providing the inputs specified by the parameter values list. We will come back to methods in the next section.

### Complex Statements

A complex statement is obtained by combining together, according to one of some possible strategies, other statements (themselves either elementary or complex). Java provides three type of strategies for building complex statements: sequencing, conditional, and iteration.

#### Sequencing

A sequencing statement is used to specify the fact that a certain set of statements has to be executed, one at a time, in the order they are written. In Java, sequencing is expressed by simply writing down the statements in the order they are meant to be executed, separated by ';' and enclosed within braces.

For example

```
{
    a = 6;
    b = 8;
    c = a*b;
}
```

is a sequencing of three (elementary) statements. This construction is also frequently known as a *block*. If one desire, it is possible to insert declaration of variables at the beginning of a

block; these variables will have a scope limited to the block and will be destroyed as soon as the block execution is completed. For example:

```
{
    int x;
    x = 2*a;
    y = x + 5;
}
```

the variable x is local to the block and will be destroyed once the execution of the block is finished. Observe that a block is viewed by Java as a single (complex) statement. Thus, we could easily write things like

```
{
    a = 6;
    {
        b = 8;
        c = a*b;
    }
    a = a+ b+c;
}
```

which is a block that contains 3 statements (the first and last are elementary, the second is a complex statement).

## Conditionals

This conditional construction allows us to combine statements with conditions, where conditions are used to decide whether the statement should be executed or not. There are two types[1] of conditionals:

```
if (condition)
    statement
```

and

```
if (condition)
    statement
else
```

[1] There is actually another type of conditional (the switch statement), but we will ignore it for the moment.

```
statement
```

In the first case (one-way condition), the specified statement will be executed only if the condition is true; in the second case (two-way condition), the condition is used to decide which of two statements to execute (the first if the condition is true, the second if false).

The conditions can be any expression that produces a result of type **boolean**.

The following are examples of conditional statements

- a simple one way condition

```
if (max < value)
{
    max = value;
    count++;
}
```

  Note that in this case the statement that is selected by the condition is a complex statement (a block).

- a two-way conditional

```
if (x > y)
    y = x;
else
{
    x++;
    y = y –x;
}
```

- a nested conditional:

```
if (x > y)
    if (z > y)
        min = y;
```

  Note that in this case the statement selected by the condition (x > y) is itself a conditional statement.

## Iterative

The iteration is used to describe the fact that a statement should be repeated a certain number of times. We will use three forms of iteration.

### while-loops

The format of this construction is

```
while (condition)
    statement
```

which requests the statement to be repeated as long as the condition is true. The condition is tested before each execution of the statement, and the statement is performed only if the condition evaluates to true.

For example

```
x = 10;
while ( x > 1)
    x = x / 2;
```

### do-while-loops

A different construction is represented by the do-while iteration; in this case, the condition of the loop is tested not before each execution of the statement, but *after* each execution. The structure is

```
do
    statement
while (condition);
```

For example,

```
do
{
    x = x-1;
    y = y * a;
}
while (x > 0);
```

### for-loops

This structure is typically used to repeat the execution of a statement for a known number of times. The structure is

```
for ( initialization  ;  condition  ;  increment )
    statement
```

The initialization and increment can be arbitrary elementary statements (or even sequences of statements separated by commas), while the condition can be any expression of type Boolean. The actual meaning of this construction can be understood by observing that it is equivalent to the construction:

```
initialization;
while (condition)
{
    statement;
    increment;
}
```

For example, if we want to repeat a statement 10 times, we can write:

```
for (i = 0;  i < 10; i++)
    statement
```

## Input and Output

In order to be able to write some simple programs, we need to know how to perform basic operations of input and output. In particular, we would like to be able to read data from the keyboard (and store them into variables) and to write things on the screen. Here we will see just some basic mechanisms to perform these operations. More sophisticated solutions to input and output will be discussed later in the course.

### Output

Java provides some methods for printing things on the screen. The two methods we will frequently use are called **print** and **println** and they are present in the class System.out. In order to be able to use them, you will need to remember to add at the top of the program the line

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

Both methods require a single argument, which is the string that should be printed on the screen. The string can be a simple sequence of characters to be copied on the screen, as in the case

```
System.out.println (“hello world”)
```

(everything within the double quotes is copied on the screen), or it can be a complex string built by concatenating different strings and values together (using the '+' operation to concatenate strings); for example:

System.out.println("the sum of "+a+" and "+b+" is "+(a+b));

The difference between print and println is that the second will also move the cursor to the beginning of the next line.

### Input

The input methods provided by Java are fairly complex. For this reason, in the first part of this course we will use a special class which provides a simplified set of input operations. The class is called SavitchIn (from the name of its author, Dr. Savitch), which provides the following methods:

- readLine () -- reads a line and returns it as a string
- readChar() -- reads a character
- readInt() -- reads the next integer
- readLong() -- reads the next long
- readDouble() -- reads the next double
- readFloat() -- reads the next float

The code for the SavitchIn class has to be stored in the same directory as the program that uses it (stored in a file called SavitchIn.java). The source code is available through the course web site.

For example, to read two numbers and computer their product:

```
System.out.print("Insert the first number: ");

a = SavitchIn.readInt();

System.out.print("Insert the second number: ");

b = SavitchIn.readInt();

System.out.println("The sum of "+a+" and "+b+" is "+ (a+b));
```

### Some Examples:

The following simple program is used to produce a some statistics about students in a class. The program is expected to read for each student a 1 if the student passed the course and a 2 if the student failed. At the end we want a count of how many students passed and how many failed. Furthermore, if more than 80% of the students passed, we want to print a special message. The input is terminated by an input equal to -1.

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

```
/**
 * Program to produce statistics about students in a class; count failures, passes
 *
 * @author Enrico Pontelli
 **/

public class gradeStatistics
{
     public static void main(String args[])
     {
          // counters for passes and failures
          int passes=0, failures=0;
          // count number of students
          int numStudents=0;
          // used for input
          int input;

          do
          {
               System.out.print("Result for next student (1=pass, 2=fail, -1=stop): ");
               input = SavitchIn.readInt();
               if (input == 1)
               {
                    passes++;
                    numStudents++;
               }
               else if (input == 2)
               {
                    failures++;
                    numStudents++;
               }
          }
          while (input != -1);
          System.out.println("Number of Failures: "+failures);
          System.out.println("Number of Successes: "+passes);
          if ( (float)passes/numStudents >=0.8)
               System.out.println("GREAT SEMESTER!");
     }
} //end class gradeStatistics
```

## More on methods

A method is used to group together a collection of statements that implement a certain functionality, assign them a name and the ability to be executed by simply specifying the name. Methods can be seen as a way to extend the language by inventing new instructions. Each new instruction performs a certain task, it has a name (the name of the method), it is capable of receiving some inputs and producing one result.

## Input Parameters

Let us start by making some observations regarding the parameters of a method. Parameters are employed to provide an input to the method at the time of its execution. Each time a method is executed, different inputs can be provided.

Let us start by looking at the following simple example

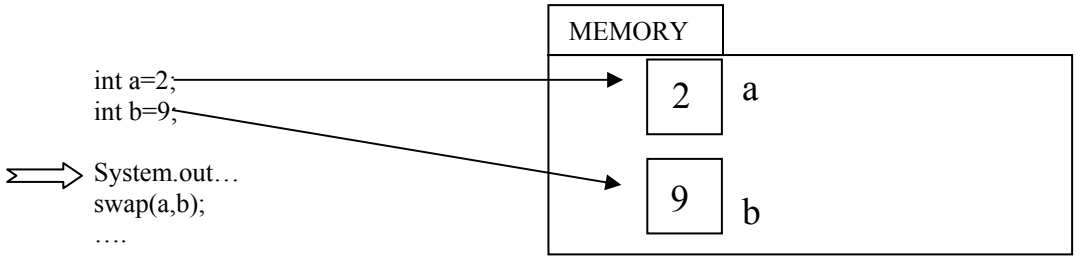
```
import java.io.*;

public class testMethod
{
    // this is a method that attempts to swap the value of two variables
    public static void swap (int x, int y)
    {
        int temp;

        temp = x;
        x = y;
        y = temp;
    }

    // this is a main program to test the method above
    public static void main (String args[])
    {
        int a=2;
        int b=9;

        System.out.println("Before swap: a="+a+" b="+b);
        swap(a,b);
        System.out.println("After swap: a="+a+" b="+b);
    }
}
```

The result of executing the program is:

Before swap: a=2 b=9

After swap: a=2 b=9

So, it appears that the two variables did not get swapped by the swap method. Why?? This effect derives from the particular methodology that Java uses to communicate the inputs to a method upon call. Let us see what really happens when we perform a call to a method. When the method **main** is started, its local variables are created and its execution starts; the memory configuration is as follows:



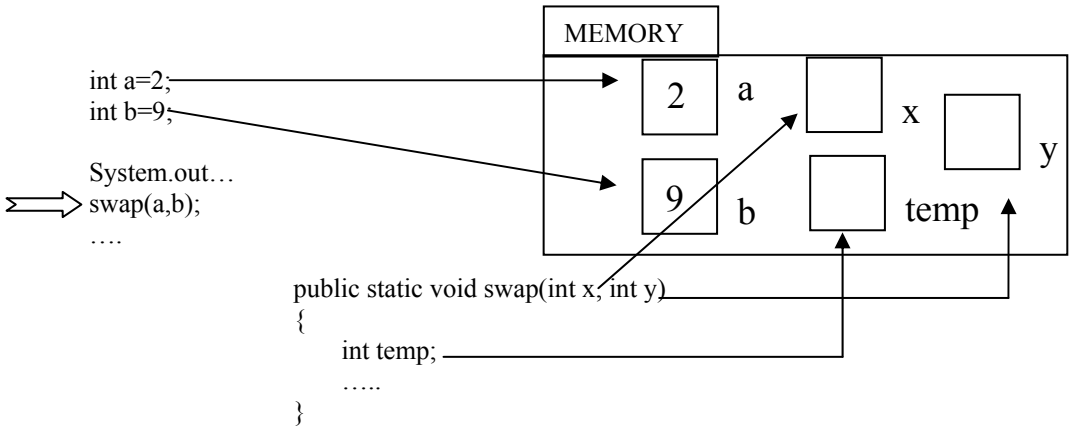

When we perform a call to a method, the following steps are taken:

- first of all, the expressions that are specified as actual parameters in the method call are completely evaluated. Each actual parameter in the call should ultimately reduce to a single element. For example, if we consider the following call to the method sqrt:

  Math.sqrt(12*x+5)

  The actual parameter in this method call is the expression 12*x+5; when the method is called, the system will first compute the value of this expression.

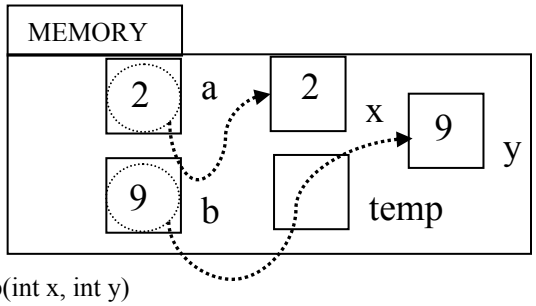
- Next, the method is started; upon starting the method, all the local variables of the method are created; in particular, the formal parameters of the method are treated just as any other local variables, and thus they are created as new variables when the method is started. Let us consider the example above. When we call the method swap, then all its local variables, including the formal parameters, are created. The memory configuration when the call starts is as follows:




- once all the local variables of the method are created, the values of the actual parameters (computed in the first step) are copied in the newly created formal

parameter variables. In our example, the values of the actual parameters (a,b) are 2,5; these values are copied in the newly created variables x,y. This is illustrated in the figure below.

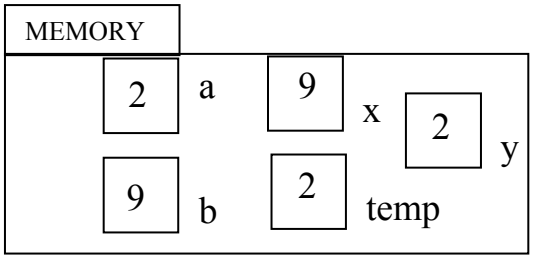
```
int a=2;
int b=9;

System.out…
⇒ swap(a,b);
….
```




```
public static void swap(int x, int y)
{
    int temp;
    …..
}
```

- after that, the method executes until the end (or until a return statement is executed). Observe that the method swap exchanges the content of its variables x and y; so at the end of the swap method, the memory configuration is as follows.

```
int a=2;
int b=9;

System.out…
⇒ swap(a,b);
….
```




```
public static void swap(int x, int y)
{
    int temp;
    …..
}
```

in particular, note that the variables a,b of the method main are not affected. This happens because the formal parameters are distinct variables than the actual parameters; they just happen to have the same values when the method starts.

So, the only thing a method receives when it start executing is a copy of the values of the actual arguments, that are stored in brand new variables created locally. All the operations that the method performs are on these local variables, and cannot affect the variables used in the actual parameters. This methodology of communication is called

**Call by Value**, since the only thing a method receives is the values of the actual parameters.

Other programming languages provide a different method of communication, called **Call by Reference**, in which the method and the caller share the actual parameters (not just their values, but the variables themselves). If Java used call by reference, then the swap method shown above would actually exchange the variables a,b (since in call by reference x,y are seen simply as new names for the variables a,b).

Java does not provide call by reference. Call by reference can be simulated by using objects, as we will discuss later in the course.

# Arrays

## Introduction

An array is a collection of variables, with the following properties:

- the collection is identified by a unique name
- the individual elements of the collection are variables; they can be accessed by specifying their position within the collection (index)
- all the variables in the collection have the same type (i.e., the collection is homogenous)
- the size of the collection is fixed (i.e., the collection is static).

## Declaring and Using Arrays

An array has to be declared before it can be used (just as any other variable). The declaration of an array has the form

data type [] array name = new data type [ array size ]

where

- data type is the type of the elements of the array
- array name is the name that we wish to give to the array
- array size is the number of elements in the collection.

For example,

```
float [] x = new float[10];
```

declares an array composed of 10 variables of type float; the array is called x. It is also possible to indicate an initial value for the elements of the array, by listing the value of each element separated by commas and within braces; e.g.,

```
float [] x = { 1.2, 3.15, 6.33, 1.6402 }
```

creates an array called x containing four elements; the first element is initialized with value 1.2, the second is assigned 3.15, etc.

The elements of the array are individual variables of type data type. They can be individually accessed by indicating the name of the array followed by the index of the element we wish to access; the index is a number between 0 and array size-1. The first element of the collection has index 0, the second has index 1, etc.

For example, if we want to double each element of an array of 10 integers:

```
for (int i=0; i<10; i++)

    array[i] = array[i]*2;
```

Java provides an attribute, automatically attached to each array, which stores the size of the array (i.e., how many elements in the collection). The attribute is called length. Thus, if x is an array, then x.length is an integer that specifies how many elements in the array x. We can easily rewrite the above code:

```
    for (int i=0; i<array.length; i++)

        array[i] = array[i] * 2;
```

## Some more considerations

### Array Objects

Let us focus a bit more on the following line:

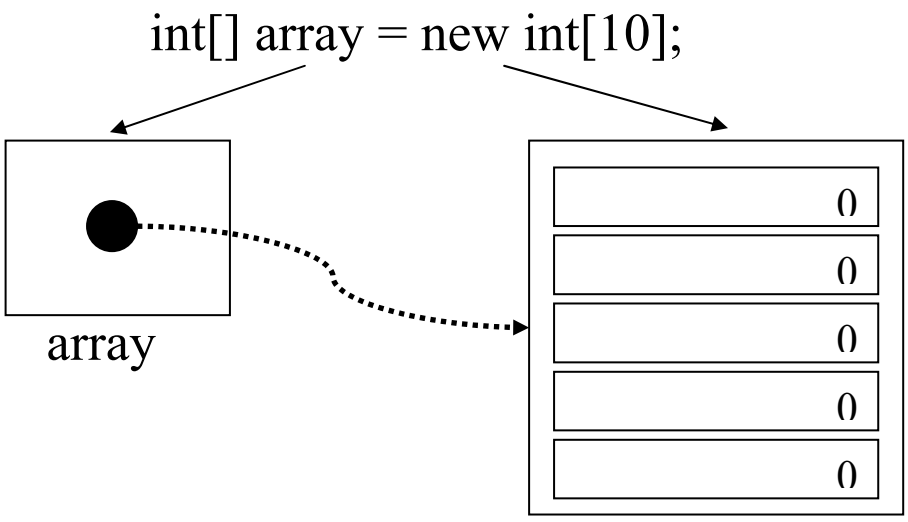
```
int[] array = new int[10];
```

what does this line really do? Two things happen when the line is executed:

1. a new variable is created; the variable is called **array**; the variable is initially empty and it is capable of containing any array of integers.

2. the part new int[10] leads to the creation of a brand new array, composed of 10 integers. The resulting entity is placed in the variable array.

This is illustrated in the figure below

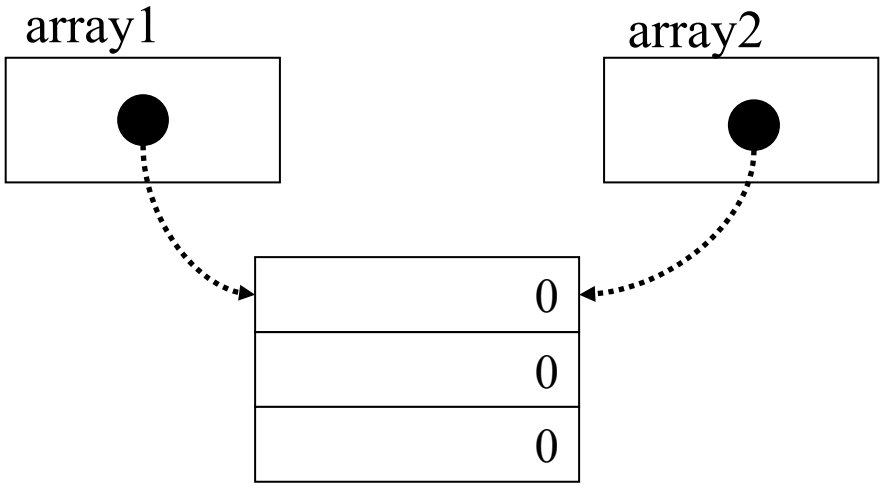



Observe that the variable called array contains a “link” to the actual array object (created by the **new** instruction). The two are separate entities; the variable array can store the “link” to any array of integers.

Consider the following example:

```
int[] array1 = new int[3];

int[] array2;

array2 = array1;
```

Note that the first line creates a variable (array1) as well as an array of 3 integers. The next line creates another variable (array2). array2 is currently empty, but it can store a link to any array of integers. The last line copies the value of array1 into array2. Since the value of array1 is the link to the array created in the first line, then this becomes also the value of array2. The resulting situation is illustrated in the following figure.




It is important to understand this behavior. In the above example, array1 and array2 are referring exactly to the same array in memory; so if one modifies the array through the variable array1, the changes will be also reflected in array2. E.g., if we execute the code:

```
for (int i=0; i<3; i++)
    array1[i] = i+1;
for (int i=0; i<3; i++)
    System.out.println("array2["+i+"]="+array2[i]);
```

what we are going to see on the screen is:

```
array2[0] = 1
array2[1] = 2
array2[2] = 3
```

So, even though the first loop modifies the content of array1, the second loop shows that the array2 has been modified as well (since array1 and array2 are really referring to the same object in memory).

### Physical Size versus Logical Size

When we create an array using the new instruction, we need to specify exactly the number of elements that the array will contain. This is what we typically call the *physical size* of the array. Such size cannot be changed – once the array is created we cannot add or remove elements from it.

There are situations where we do not know a priori how many elements we will need to store in the array. For example, we might be reading the grades for a class, we want to store those grades in an array, but we have no clue of how many students are there. Since the array has to

be crated before we start reading the grades, we will need to estimate the size and hope for the best. For example, from previous experience we might believe that there are going to be at most 50 students, so we proceed to create an array with 50 grades. If in practice we end up with less than 50 grades (e.g., 25), then we will use only a part of the array. E.g.,

```
int[] grades = new int[50];
int count_grades = 0;
int next;

do
{
    System.out.print("Next grade: (-1 to stop) ");
    next = SavitchIn.readInt();
    if (next != -1)
        grades[count_grades++] = next;
}
while (next != -1);
```

In this example, 50 is the physical size of the array, while the value of the variable count_grades at the end of the loop will indicate the *logical size* of the array – i.e., how many elements of the array we are actually using.

What shall we do if the physical size turns out to be less than the logical size? We need in this case to "grow" the array to make space for the extra elements. Unfortunately there is no easy way to grow an array in Java. The only approach is to:

- create a new, bigger array
- copy all the elements from the old array to the new one
- forget about the old array and continue using the new one

This is illustrated in the following modified version of our code to read and store the grades:

```
int[] grades = new int[50];
int count_grades = 0;
int next;

do
{
    System.out.print("Next grade: (-1 to stop) ");
    next = SavitchIn.readInt();
    if (next != -1)
    {
        // let us check if we have space for the new element
        if (count_grades < grades.length)
            grades[count_grades++] = next;
        else
        {
            // array is full; we need to grow it; let's replace it with one twice as big
```

```
                int[] newarray = new int[ grades.length * 2];

                // copy values from old array to the new one
                for (int j=0; j < grades.length; j++)
                     newarray[j] = grades[j];

                // let's now replace the grades array with the new one
                grades = newarray;

                // and finally insert the new value
                grades[count_grades++] = next;
           }
}
while (next != -1);
```

Observe that once we have prepared the new array (called newarray in the example), we simply store a link to it in the variable grades. At this point, the value of the variable grades is a link to the newly created, bigger array. What happened to the old array? It is probably still somewhere in memory, but it is not linked by the variable grades any longer (it is "unreachable"). This is what we typically call *garbage*. An object in memory becomes garbage when there are no more variables that are linking to it (i.e., nobody is referring to this object any longer). Java periodically runs a particular algorithm, called *garbage collection*, which analyzes the content of memory to detect garbage. Any garbage item is recalled and its memory reused for other purposes.

Some Examples

## Computing Average Grade

Consider the problem of computing the average grade in a class. The simplest version of this program simply reads a number of grades and computes the average by summing the grades together. The following program assumes that there are exactly 20 students in the class:

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

class AverageOf20
{
    public static void main( String args[] )
    {
        int sum;        // this will be the running sum
        int grade;      // this will be the current grade (just read from the user)
        double avg;     // this will contain the average grade in the class (note the type double)

        // loop to read 20 grades
        for (int  i = 0; i < 20; i++)
        {
            // ask for a grade
            System.out.print("Please insert the next grade: ");
            grade =SavitchIn.readInt();
            // add the grade to the running sum
            sum = sum + grade;
        }
        // compute the average
        avg = sum / 20.0;          // note the use of 20.0 instead of 20; required to force a real division
        // display the result
        System.out.println("The average grade is "+avg);
    }
}
```

Please pay attention to the comments in the program, as they highlight some features of the program.

Let us now generalize the program to request the ability to work with any number of grades (not necessarily 20). In this case we will ask the user to type a special value to indicate the end of the input (this is what we typically call a *sentinel-driven loop*, where the sentinel is the special value that will terminate the loop).

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

```
class AverageOfAny
{
	public static void main( String args[] )
	{
		int sum=0;		// this will be the running sum
		int grade=0;		// this will be the current grade (just read from the user)
		int count = 0;		// used to count how many grades have been read
		double avg;	// this will contain the average grade in the class (note the type double)

		// loop to read grades; the value -1 is the sentinel
		while (grade != -1)
		{
			// ask for a grade
			System.out.print("Please insert the next grade: ");
			grade = SavitchIn.readInt();
			// we need to check the grade to make sure it's not the sentinel
			if (grade != -1)
			{
				// add the grade to the running sum
				sum = sum + grade;
				// increment the counter of the grades
				count++;
			}
		}
		// compute the average; need to make sure that the count of grades is not zero
		if (count == 0)
			avg = 0.0;
		else
			avg = (double)sum / count;		// note the type cast to force a real division
		// display the result
		System.out.println("The average grade is "+avg);
	}
}
```

Now let us extend the program so that we can also determine the maximum grade. For the sake of exercise, we will use arrays to store all the grades, and we will define a method to compute the maximum value inside an array. Again, for simplicity we will fix the number of grades to 20.

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

```
import java.lang.*;

class AverageOfAny
{
	public static int maximum (int array[])
	{
		int max = array[0];		// initial guess: the first element is the max

		// compare it with all other elements
		for (int i = 1; i < array.length; i++)
		{
			if (max < array[i])
				max = array[i];		// found a greater element; update the max
		}
		// send back the maximum
		return (max);
	}

	public static void main( String args[] )
	{
		int sum=0;				// this will be the running sum
		int[] grades=new int[20];		// this will store all the 20 grades
		double avg;	// this will contain the average grade in the class (note the type double)

		// loop to read grades;
		for (int i=0; i<20; i++)
		{
			// ask for a grade
			System.out.print("Please insert the next grade: ");
			grades[i] =SavitchIn.readInt();
			// add the grade to the running sum
			sum = sum + grades[i];
		}
		// compute the average; need to make sure that the count of grades is not zero
		avg = (double)sum /20;		// note the type cast to force a real division
		// display the result
		System.out.println("The average grade is "+avg);
		// now let's print the maximum grade
		System.out.println("The highest grade in the class is: "+maximum(grades));
	}
```

```
}
```

## Guess the number game

The following program plays a simple game of guessing the number; the program will generate a random number and the user will have a number of attempts to guess the number. Methods and global data are used in this program.

```
import java.io.*;
import java.lang.*;
import java.util.*;
class Game
{
    static int Number;          // the number to be guessed
    public static void generateNumber()
    {
        Random r = new Random();
        Number =r.nextInt(100) + 1;   // generate random number between 1 and 100
    }
    public static boolean checkNumber (int guess)
    {
        if (guess == Number)
        {
            System.out.println("WINNER!!");
            return (true);
        }
        else if (guess < Number)
        {
            System.out.println("Too small!");
            return (false);
        }
        else
        {
            System.out.println("Too big!");
            return (false);
        }
    }

    public static void main (String args[])
    {
        int guess;
        generateNumber();
        do
        {
            System.out.print("Your guess: ");
            guess = SavitchIn.readInt();
        }
        while (! checkNumber(guess));
```

```
    }
}
```