Control Structures

A program is usually not limited to a linear sequence of instructions. During its process it may bifurcate, repeat code or take decisions. For that purpose, Java provides control structures that serve to specify what has to be done by our program, when and under which circumstances.

With the introduction of control structures we are going to have to introduce a new concept: the *compound-statement* or *block*. A block is a group of statements which are separated by semicolons (**;**) like all Java statements, but grouped together in a block enclosed in braces: { }:

{ statement1; statement2; statement3; }

Most of the control structures require a generic statement as part of its syntax. A statement can be either a simple statement (a simple instruction ending with a semicolon) or a compound statement (several instructions grouped in a block), like the one just described. In the case that we want the statement to be a simple statement, we do not need to enclose it in braces ({}). But in the case that we want the statement to be a compound statement it must enclosed between braces ({}) forming a block.

Conditional Structure: if and else

The **if** keyword is used to execute a statement or block only if a condition is fulfilled. Its form is:

if (*condition*) statement

Where ***condition*** is the expression that is being evaluated. If this ***condition*** is true, ***statement*** is executed. If it is false, ***statement*** is ignored (not executed) and the program continues right after this conditional structure.

For example, the following code fragment prints **x is 100** only if the value stored in the x variable is indeed 100:

```
if (x == 100)
  System.out.print("x is 100");
```

If we want more than a single statement to be executed in case that the condition is true we can specify a block using braces { }:

```
if (x == 100)
{
   System.out.print("x is ");
   System.out.print(x);
}
```

We can additionally specify what we want to happen if the condition is not fulfilled by using the keyword else. Its form used in conjunction with if is:

if (*condition*) statement1 else statement2

For example:

```
if (x == 100)
  System.out.print("x is 100");
else
  System.out.print("x is not 100");
```

prints on the screen **x is 100** if indeed x has a value of 100, but if it has not -and only if not- it prints out **x is not 100**.

The if + else structures can be concatenated with the intention of verifying a range of values. The following example shows its use telling if the value currently stored in x is positive, negative or none of them (i.e. zero):

```
if (x > 0)
  System.out.print("x is positive");
else if (x < 0)
  System.out.print("x is negative");
else
  System.out.print("x is 0");
```

Remember that in case that we want more than a single statement to be executed, we must group them in a block by enclosing them in braces { }.

The Selective Structure: switch.

The syntax of the switch statement is a bit peculiar. Its objective is to check several possible constant values for an expression. Something similar to what we did at the beginning of this lesson with the concatenation of several if and else if instructions. Its form is the following:

```
switch (expression)
{
  case constant1:
     group of statements 1;
     break;
  case constant2:
     group of statements 2;
     break;
  .
  .
  default:
     default group of statements
}
```

It works in the following way: switch evaluates expression and checks if it is equivalent to constant1, if it is, it executes group of statements 1 until it finds the break statement. When it finds this break statement the program jumps to the end of the switch selective structure.

If expression was not equal to constant1 it will be checked against constant2. If it is equal to this, it will execute group of statements 2 until a break keyword is found, and then will jump to the end of the switch selective structure.

Finally, if the value of expression did not match any of the previously specified constants (you can include as many case labels as values you want to check), the program will execute the statements included after the default: label, if it exists (since it is optional).

Both of the following code fragments have the same behavior:

switch example	if-else equivalent

`switch (x)`
`{`
`  case 1:`
`  System.out.print("x is 1");`
`  break;`
`  case 2:`
`  System.out.print("x is 2");`
`  break;`
`  default:`
`  System.out.print("x is unknown");`
`}`
	`if (x == 1)`
	`{`
	`  System.out.print("x is 1");`
	`}`
	`else if (x == 2)`
	`{`
	`  System.out.print("x is 2");`
	`}`
	`else`
	`{`
	`  System.out.print("x is unknown";`
	`}`

The switch statement is a bit peculiar within the Java language because it uses labels instead of blocks. This forces us to put break statements after the group of statements that we want to be executed for a specific condition. Otherwise the remainder statements - including those corresponding to other labels - will also be executed until the end of the switch selective block or a break statement is reached.

For example, if we did not include a break statement after the first group for case one, the program will not automatically jump to the end of the switch selective block and it would continue executing the rest of statements until it reaches either a break instruction or the end of the switch selective block. This makes unnecessary to include braces { } surrounding the statements for each of the cases, and it can also be useful to execute the same block of instructions for different possible values for the expression being evaluated. For example:

```
switch (x)
{
  case 1:
  case 2:
  case 3:
    System.out.print("x is 1, 2 or 3");
    break;
  default:
    System.out.print("x is not 1, 2 nor 3");
}
```

Notice that switch can only be used to compare an expression against constants. Therefore we cannot put variables as labels (for example case n: where n is a variable) or ranges (case (1..3):) because they are not valid C++ constants.

If you need to check ranges or values that are not constants, use a concatenation of if and else if statements.

Iteration Structures (loops)

Loops have as purpose to repeat a statement a certain number of times or while a condition is fulfilled.

The while loop

Its format is:

while (expression) statement

and its functionality is simply to repeat statement while the condition set in expression is true. For example, we are going to make a program to count down using a while-loop:

```
// custom countdown using while

public static void main(String args[])
{
  int n;
  Sustem.out.println("Enter the starting number > ";
  n = (new Scanner(System.in)).nextInt();

  while (n>0)
  {
    System.out.print(n + ", ");
    --n;
  }
  System.out.print("FIRE!");
}
```

```
Enter the starting number > 8

8, 7, 6, 5, 4, 3, 2, 1, FIRE!
```

When the program starts the user is prompted to insert a starting number for the countdown. Then the while loop begins, if the value entered by the user fulfills the condition **n>0** (that n is greater than zero) the block that follows the condition will be executed and repeated while the condition (**n>0**) remains being true.

The whole process of the previous program can be interpreted according to the following script (beginning in main):

1. User assigns a value to n
2. The while condition is checked (n>0). At this point there are two possibilities:
   - condition is true: statement is executed (to step 3)
   - condition is false: ignore statement and continue after it (to step 5)
3. Execute statements:
   System.out.print(n + ", ");
   --n;
   (prints the value of n on the screen and decreases n by 1)
4. End of block. Return automatically to step 2

5. Continue the program right after the block: print FIRE! and end program.

When creating a while-loop, we must always consider that it has to end at some point, therefore we must provide within the block some method to force the condition to become false at some point, otherwise the loop will continue looping forever. In this case we have included --n; that decreases the value of the variable that is being evaluated in the condition (n) by one - this will eventually make the condition (n>0) to become false after a certain number of loop iterations: to be more specific, when n becomes 0, that is where our while-loop and our countdown end.

Of course this is such a simple action for our computer that the whole countdown is performed instantly without any practical delay between numbers.

The for loop

Its format is:

for (*initialization; condition; increase*) statement;

and its main function is to repeat statement while condition remains true, like the while loop. But, in addition, the for loop provides specific locations to contain an initialization statement and an increase statement. So this loop is specially designed to perform a repetitive action with a counter which is initialized and increased on each iteration.
It works in the following way:

1. initialization is executed. Generally it is an initial value setting for a counter variable. This is executed only once.
2. condition is checked. If it is true the loop continues, otherwise the loop ends and statement is skipped (not executed).
3. statement is executed. As usual, it can be either a single statement or a block enclosed in braces { }.
4. finally, whatever is specified in the increase field is executed and the loop gets back to step 2.

Here is an example of countdown using a for loop:

```
// countdown using a for loop
public static void main(String args[])
{
  for (int n=10; n>0; n--) {
    System.out.print(n + ", ");
  }
  System.out.print("FIRE!");
}
```

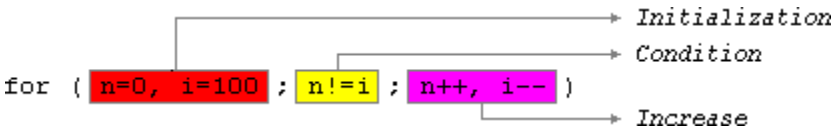
```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FIRE!
```

The initialization and increase fields are optional. They can remain empty, but in all cases the semicolon signs between them must be written. For example we could write: for (;n<10;) if we wanted to specify no initialization and no increase; or for (;n<10;n++) if we wanted to include an increase field but no initialization (maybe because the variable was already initialized before).

Optionally, using the comma operator (,) we can specify more than one expression in any of the fields included in a for loop, like in initialization, for example. The comma operator (,) is an expression separator, it serves to separate more than one expression where only one is generally expected. For example, suppose that we wanted to initialize more than one variable in our loop:

```
for ( n=0, i=100 ; n!=i ; n++, i-- )
{
   // whatever here...
}
```

This loop will execute for 50 times if neither n or i are modified within the loop:



n starts with a value of 0, and i with 100, the condition is n!=i (that n is not equal to i). Because n is increased by one and i decreased by one, the loop's condition will become false after the 50th loop, when both n and i will be equal to 50.

Branching Structures: Methods and Classes

Methods

Using methods we can structure our programs in a more modular way, accessing all the potential that structured programming can offer to us in Java.

A method is a group of statements that is executed when it is called from some point of the program. The following is its format:

access_modifier type name (*argument1, argument2,* ...) { statement }

where:

- **access_modifier** specifies how restricted the access is to a class or interface or variable or method
- **type** is the data type specifier of the data returned by the method.
- **name** is the identifier by which it will be possible to call the method.
- **arguments** (as many as needed): Each argument consists of a data type specifier followed by an identifier, like any regular variable declaration (for example: int x) and which acts within the method as a regular local variable. They allow the passing of arguments to the method when it is called. The different arguments are separated by commas.
- **statements** is the method's body. It is a block of statements surrounded by braces { }.

Here you have the first method example:

```
public static int addition(int a, int b)
{
  int r;
  r=a+b;
  return r;
}

public static void main(String args[])
{
  int z;
  z = addition (5,3);
  System.out.print("The result is ");
}
```

```
The result is 8
```

In order to examine this code, first of all remember a Java program always begins its execution by the main method. So we will begin there.

We can see how the main method begins by declaring the variable z of type int. Right after that, we see a call to a method called addition. Paying attention we will be able to see the similarity between the structure of the call to the method and the declaration of the method itself:

```
int addition (int a, int b)
              ↑      ↑
z = addition (  5   ,   3  );
```

There is a clear correspondence between the arguments in the method call and the arguments in the method declaration. Within the main method we called to addition passing two values: 5 and 3, that correspond to the int a and int b arguments declared for method addition.

At the point at which the method is called from within main, the control is lost by main and passed to method addition. The value of both arguments passed in the call (5 and 3) are copied to the local variables int a and int b within the method.

Method addition declares another local variable (int r), and by means of the expression r=a+b, it assigns to r the result of a plus b. Because the actual argumentss passed for a and b are 5 and 3 respectively, the result is 8.

The following line of code:

```
return r;
```

finalizes method addition, and returns the control back to the method that called it in the first place (in this case, main). At this moment the program follows it regular course from the same point at which it was interrupted by the call to addition. But additionally, because the return statement in method addition specified a value: the content of variable r (return r;), which at that moment had a value of 8. This value becomes the value of evaluating the method call.

```
int addition (int a, int b)
 |8
z = addition (  5   ,   3  );
```

So being the value returned by a method the value given to the method call itself when it is evaluated, the variable z will be set to the value returned by addition(5, 3), that is 8. To explain it another way, you can imagine that the call to a method (addition(5,3)) is literally replaced by the value it returns (8).

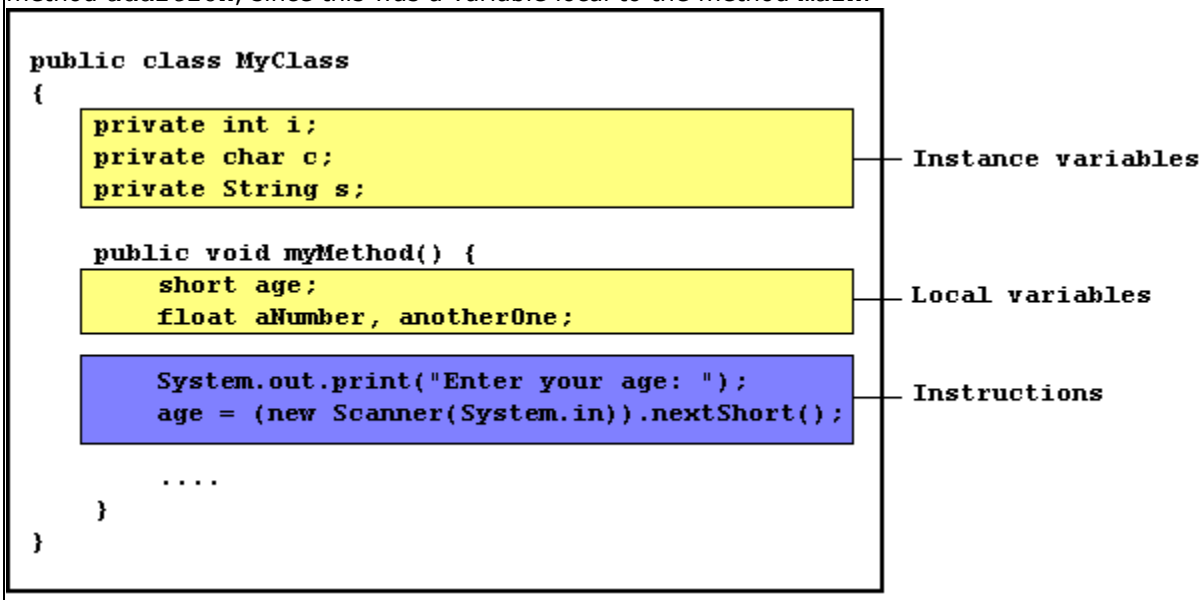
The following line of code in main is:

```
System.out.print("The result is " + z);
```

That, as you may already expect, produces the printing of the result on the screen.

Scope of variables

The scope of variables declared within a method or any other inner block is only their own method or their own block and cannot be used outside of them. For example, in the previous example it would have been impossible to use the variables `a`, `b` or `r` directly in method `main` since they were variables local to method `addition`. Also, it would have been impossible to use the variable `z` directly within method `addition`, since this was a variable local to the method `main`.



Therefore, the scope of local variables is limited to the same block level in which they are declared. Nevertheless, we also have the possibility to declare instance variables; These are visible from any point of the code, inside and outside all methods. In order to declare instance variables you simply have to declare the variable outside any method or block; that means, directly in the body of the class.

And here is another example about methods:

```
public int subtraction (int a, int b)
{
  int r;
  r=a-b;
  return (r);
}

public void someMethod()
{
  int x=5, y=3, z;
  z = subtraction (7,2);
  System.out.println("The first result is " + z);
  System.out.println("The second result is " +
         subtraction(7,2));
  System.out.println("The third result is " +
         Subtraction(x,y));
  z= 4 + subtraction(x,y);
  System.out.println("The fourth result is " + z);
}
```

```
The first result is 5
The second result is 5
The third result is 2
The fourth result is 6
```

In this case we have created a method called subtraction. The only thing that this method does is to subtract both passed arguments and to return the result.

Nevertheless, if we examine method main we will see that we have made several calls to method subtraction. We have used some different calling methods so that you see other ways or moments when a method can be called.

In order to understand well these examples you must consider once again that a call to a method could be replaced by the value that the method call itself is going to return. For example, the first case (that you should already know because it is the same pattern that we have used in previous examples):

```
z = subtraction (7,2);
System.out.println("The first result is " + z);
```

If we replace the method call by the value it returns (i.e., 5), we would have:

```
z = 5;
System.out.println("The first result is " + z);
```

As well as

```
System.out.println("The second result is " + subtraction (7,2));
```

has the same result as the previous call, but in this case we made the call to subtraction directly as part of the argument for System.out.println. Simply consider that the result is the same as if we had written:

```
System.out.println("The second result is " + 5);
```

since 5 is the value returned by subtraction(7,2).

In the case of:

```
System.out.println("The third result is " + subtraction(x,y));
```

The only new thing that we introduced is that the argumentss of subtraction are variables instead of constants. That is perfectly valid. In this case the values passed to method subtraction are the values of x and y, that are 5 and 3 respectively, giving 2 as result.

The fourth case is more of the same. Simply note that instead of:

```
z = 4 + subtraction(x,y);
```

we could have written:

```
z = subtraction(x,y) + 4;
```

with exactly the same result. I have switched places so you can see that the semicolon sign (;) goes at the end of the whole statement. It does not necessarily have to go right after the method call. The explanation might be once again that you imagine that a method can be replaced by its returned value:

```
z = 4 + 2;
z = 2 + 4;
```

Methods With No Return Type. The Use Of void.

If you remember the syntax of a method declaration:

access_modifier type name (*argument1*, *argument2* ...) statement

you will see that the declaration includes a type, that is the type of the method itself (i.e., the type of the datum that will be returned by the method with the return statement). But what if we want to return no value?

Imagine that we want to make a method just to show a message on the screen. We do not need it to return any value. In this case we should use the void type specifier for the method. This is a special specifier that indicates absence of type.

```
public void printmessage()
{
  System.out.print("I'm a method!");
}

public void someMethod()
{
  printmessage();
}
```

```
I'm a method!
```

What you must always remember is that the format for calling a method includes specifying its name and enclosing its arguments between parentheses. The non-existence of arguments does not exempt us from the obligation to write the parentheses. For that reason the call to printmessage is:

```
printmessage();
```

The parentheses clearly indicate that this is a call to a method and not the name of a variable or some other Java statement. The following call would have been incorrect:

```
printmessage;
```

Classes

A **class** is an expanded concept of a data structure: instead of holding only data, it can hold both data and methods.

An **object** is an instantiation of a class. In terms of variables, a class would be the type, and an object would be the variable.

Classes are declared using the keyword class, with the following format:

```
access_modifier class class_name
{

}
```

Where class_name is a valid identifier for the class. The body of the declaration can contain members, that can be either data or method declarations, and optionally access specifiers.

An access modifier is one of the following three keywords: *private*, *public* or *protected*. These specifiers modify the access rights of a class or class member:

- The *public* access modifier is the least restrictive. Any thing declared as *public* can be accessed from any class. Outer classes are usually declared as *public* so they can be instantiated from other classes.

- The *private* access modifier is the most restrictive of the access modifiers. A private member is only available to its own class. Thus, an outer class is not declared as private, because it could not be instantiated by any other class. If a method is *private*, only other methods from that class can invoke the private method. Using private for methods that assist other methods but never could be of use for other classes by themselves is a wise idea. Outer classes can not be declared private.
- The *protected* access modifier is the first to use packages. A *protected* member is available to other classes in the same package and also to any subclass regardless of which package the subclass is in. Classes cannot be declared *protected*.
- The default modifier, nicknamed "friendly", is inappropriately named. It would seem that "friendly" should be close to *public* in its meaning, but it is actually on the other side of the spectrum. The "friendly" setting is the most restrictive access modifier after *private*. A "friendly" member or class is available to other classes in the same package. It is different from protected, because a subclass that resides in another package does not have access to a protected member of the parent class. If you are wondering why the nickname "friendly" is used to characterize something that is not too friendly, that makes two of us.

One of the greatest advantages of a class is that, as any other type, we can declare several objects of it. For example, if we declare a class Rectangle, we could declared the object rectA and the object rectB:

```
public class Rectangle
{
    private int x, y;

    public void setValues(int a,int b) {
        x = a;
        y = b;
    }

    public int area() {
        return (x*y);
    }
}

public class MyClass
{
    public static void main (String args[]) {
        Rectangle rectA = new Rectangle();
        Rectangle rectB = new Rectangle();
        rectA.set_values (3,4);
        rectB.set_values (5,6);
        System.out.println("rectA area: " + rectA.area());
        System.out.println("rectB area: " + rectB.area());
    }
}
```

```
rect area: 12
rectb area: 30
```

In this concrete case, the class (type of the objects) to which we are talking about is **Rectangle**, of which there are two instances or objects: **rectA** and **rectB**. Each one of them has its own member variables and member methods.

Notice that the call to **rectA.area()** does not give the same result as the call to **rectB.area()**. This is because each object of class Rectangle has its own variables **x** and **y**, as they, in some way, have also their own member methods **setValue()** and **area()** that each uses its object's own variables to operate.

That is the basic concept of object-oriented programming: Data and methods are both members of the object. We no longer use sets of global variables that we pass from one method to another as arguments, but instead we handle objects that have their own data and methods embedded as members. Notice that we have not had to give any arguments in any of the calls to rectA.area or rectB.area. Those member methods directly used the data members of their respective objects rectA and rectB.

Constructors

Objects generally need to initialize variables or assign dynamic memory during their process of creation to become operative and to avoid returning unexpected values during their execution. For example, what would happen if in the previous example we called the member method **area()** before having called method **setValues()**? Probably we would have gotten an undetermined result since the members x and y would have never been assigned a value.

In order to avoid that, a class can include a special method called a constructor, which is automatically called whenever a new object of this class is created. This constructor method must have the same name as the class, and cannot have any return type; not even void.

We are going to implement the Rectangle class including a constructor:

```
public class Rectangle {
    private int width, height;

    public Rectangle(int a, int b) {
        width = a;
        height = b;
    }

    public int area()
        return (width*height);
    }
}

public class MyClass
{
    public static void main(String args[]) {
        Rectangle rectA = new Rectangle(3,4);
        Rectangle rectB = new Rectangle(5,6);
        System.out.println("rectA area: " + rectA.area());
        System.out.println("rectB area: " + rectb.area());
    }
}
```

```
rectA area: 12
rectB area: 30
```

As you can see, the result of this example is identical to the previous one. But now we have removed the member method **setValues()**, and have included instead a constructor that performs a similar action: it initializes the values of x and y with the arguments that are passed to it.

Notice how these arguments are passed to the constructor at the moment at which the objects of this class are created:

```
Rectangle rectA = new Rectangle(3,4);
Rectangle rectB = new Rectangle(5,6);
```

Constructors cannot be called explicitly as if they were regular member methods. They are only executed when a new object of that class is created.

You can also see how neither the prototype nor the later constructor declaration includes a return value; not even void.

Overloading Constructors

Like any other method, a constructor can also be overloaded with more than one method that has the same name but different types or number of arguments. Remember that for overloaded methods the compiler will call the one whose arguments match the arguments used in the method call. In the case of constructors, which are automatically called when an object is created, the one executed is the one that matches the arguments passed on the object declaration:

```
public class Rectangle
{
    private int width, height;

    public Rectangle() {
        width = 5;
        height = 5;
    }

    public Rectangle(int a, int b) {
        width = a;
        height = b;
    }

    public int area () {
        return (width*height);
    }
}

public class MyClass
{
    public static void main(String args[]) {
        Rectangle rectA = new Rectangle(3,4);
        Rectangle rectB = new Rectangle();
        System.out.println("rectA.area: " + rectA.area());
        System.out.println("rectB.area: " + rectb.area());
    }
}
```

```
rect area: 12
rectb area: 25
```

In this case, rectB was declared without any arguments, so it has been initialized with the constructor that has no argumentss, which initializes both width and height with a value of 5.

Default Constructor

If you do not declare any constructors in a class definition, the compiler assumes the class to have a default constructor with no arguments. Therefore, after declaring a class like this one:

```
public class Example
{
    private int a,b,c;

    public void multiply(int n, int m) {
        a=n;
        b=m;
        c=a*b;
    }
}
```

The compiler assumes that **Example** has a default constructor, so you can declare objects of this class by simply declaring them without any arguments:

```
Example ex = new Example();
```

But as soon as you declare your own constructor for a class, the compiler no longer provides an implicit default constructor. So you have to declare all objects of that class according to the constructor prototypes you defined for the class:

```
public class Example
{
  private int a,b,c;

    public Example(int n, int m) {
        a=n;
        b=m;
    }
    public void multiply() {
        c=a*b;
    }
}
```

Here we have declared a constructor that takes two arguments of type int. Therefore the following object declaration would be correct:

```
Example ex = new Example(2,3);
```

But,

```
Example ex = new Example();
```

Would not be correct, since we have declared the class to have an explicit constructor, thus replacing the default constructor.

Static Members

A class can contain static members, either data or methods.

Static data members of a class are also known as "class variables", because there is only one unique value for all the objects of that same class. Their content is not different from one object of this class to another.

For example, it may be used for a variable within a class that can contain a counter with the number of objects of that class that have been created, as in the following example:

```
public class Dummy
{
    public static int n;

    Dummy () {
        n++;
    }
}

public class MyClass
{
    public static void main(String args[]) {
        Dummy a = new Dummy();
        Dummy b = new Dummy();;
        Dummy c = new Dummy();
        System.out.println(a.n);
    }
}
```

```
3
```

In fact, static members have the same properties as instance variables - they enjoy class scope.

Because it is a unique variable value for all the objects of the same class, it can be referred to as a member of any object of that class or even directly by the class name (of course this is only valid for public static members):

```
System.out.print(a.n);
System.out.print(Dummy.n);
```

These two calls included in the previous example are referring to the same variable: the *static* variable *n* within class *Dummy* shared by all objects of this class.

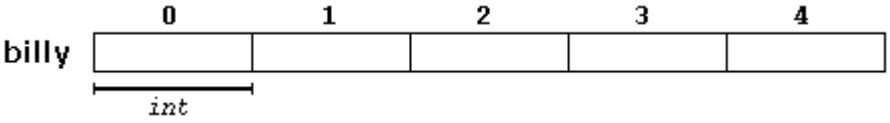
Just as we may include static data within a class, we can also include static methods. They represent the same: they are class methods that are called as if they were object members of a given class. They can only refer to static data, in no case to non-static members of the class, as well as they do not allow the use of the keyword *this*, since it makes reference to an object pointer and these methods in fact are not members of any object but direct members of the class.

Arrays

An array is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier.

That means that, for example, we can store 5 values of type int in an array without having to declare 5 different variables, each one with a different identifier. Instead of that, using an array we can store 5 different values of the same type, int for example, with a unique identifier.

For example, an array to contain 5 integer values of type int called billy could be represented like this:



where each blank panel represents an element of the array, that in this case are integer values of type int. These elements are numbered from 0 to 4 since in arrays the first index is always 0, independently of its length.

Like a regular variable, an array must be declared before it is used. A typical declaration for an array in Java is:

type[] name = new type[elements];

where type is a valid type (like *int*, *float*...), name is a valid identifier and the elements field (which is always enclosed in square brackets []), specifies how many of these elements the array has to contain.
Therefore, in order to declare an array called billy as the one shown in the above diagram it is as simple as:

```
int[] billy = new int[5];
```

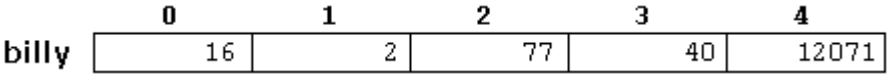
Initializing Arrays

When declaring an array, if we do not specify otherwise, its elements will all be automatically initialized filled with zeros.

When we declare an array, we have the possibility to assign initial values to each one of its elements by enclosing the values in braces { }. For example:

```
int[] billy = { 16, 2, 77, 40, 12071 };
```

This declaration would have created an array like this:



When an initialization of values is provided for an array, Java requires that the square brackets remain empty [ ]. In this case, the compiler will assume a size for the array that matches the number of values included between braces { }:

```
int billy [] = { 16, 2, 77, 40, 12071 };
```

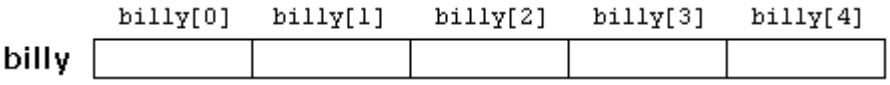
After this declaration, array billy would be 5 ints long, since we have provided 5 initialization values.

Accessing The Values In An Array

In any point of a program in which an array is visible, we can access the value of any of its elements individually as if it was a normal variable, thus being able to both read and modify its value. The format is as simple as:

name[index]

Following the previous examples in which billy had 5 elements and each of those elements was of type int, the name which we can use to refer to each element is the following:



For example, to store the value 75 in the third element of billy, we could write the following statement:

```
billy[2] = 75;
```

and, for example, to pass the value of the third element of billy to a variable called a, we could write:

```
a = billy[2];
```

Therefore, the expression billy[2] is for all purposes like a variable of type int.

Notice that the third element of billy is specified billy[2], since the first one is billy[0], the second one is billy[1], and therefore, the third one is billy[2]. By this same reason, its last element is billy[4]. Therefore, if we write billy[5], we would be accessing the sixth element of billy and therefore exceeding the size of the array.

In Java it is syntactically correct to exceed the valid range of indices for an Array. This can create problems, since accessing out-of-range elements do not cause compilation errors but can cause runtime errors.

At this point it is important to be able to clearly distinguish between the two uses that brackets [ ] have related to arrays. They perform two different tasks: one is to specify the size of arrays when they are declared; and the second one is to specify indices for concrete array elements. Do not confuse these two possible uses of brackets [ ] with arrays.

```
int[] billy = new int[5];      // declaration of a new Array
billy[2] = 75;                 // access to an element of the Array.
```

If you read carefully, you will see that a type specifier always precedes a variable or array declaration, while it never precedes an access.

Some other valid operations with arrays:

```
billy[0] = a;
billy[a] = 75;
b = billy[a+2];
billy[billy[a]] = billy[2] + 5;
```

jimmy

	0	1	2	3	4
0	1	2	3	4	5
1	2	4	6	8	10
2	3	6	9	12	15

Arrays As Arguments

At some moment we may need to pass an array to a method as an argument. In Java it is not possible to pass a complete block of memory by value as an argument to a method, but we are allowed to pass its address. In practice this has almost the same effect and it is a much faster and more efficient operation.

In order to accept arrays as arguments the only thing that we have to do when declaring the method is to specify in its arguments the element type of the array, an identifier and a pair of void brackets []. For example, the following method:

```
public void procedure (int arg[])
```

accepts an argument of type "Array of int" called arg. In order to pass to this method an array declared as:

```
int[] myarray = new int[40];
```

it would be enough to write a call like this:

```
procedure(myarray);
```

Here you have a complete example:

```
public void printarray (int arg[], int length) {
  for (int n=0; n<length; n++)
    System.out.print(arg[n] + " ");
  System.out.println();
}

public void someMethod()
{
  int firstarray[] = {5, 10, 15};
  int secondarray[] = {2, 4, 6, 8, 10};
  printarray (firstarray,3);
  printarray (secondarray,5);
}
```

```
5 10 15
2 4 6 8 10
```

As you can see, the first argument (int arg[]) accepts any array whose elements are of type int, whatever its length. For that reason we have included a second argument that tells the method the length of each array that we pass to it as its first argument. This allows the for loop that prints out the array to know the range to iterate in the passed array without going out of range.

LAB 22 - ASSIGNMENT

Lab 23A - 60 points

OBJECTIVE

WAP that reads the radius and the height of a cone from the keyboard. Calculate and display the surface area and volume of the cone.

Cone

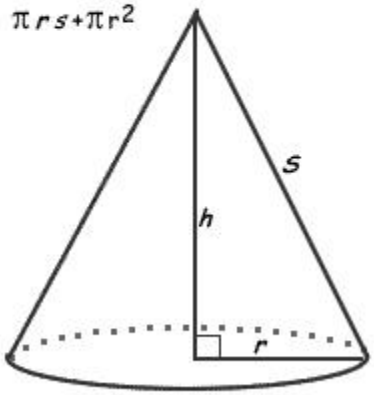
Surface Area

We will need to calculate the surface area of the cone and the base.

Area of the cone is πrs

Area of the base is πr^2

Therefore the Formula is: $SA = \pi rs + \pi r^2$



Volume

$V = \frac{1}{3} \pi r^2 h$

FIELD SUMMARY

- **int radius** – the radius of the base of the cone.
- **int height** – the height of the cone.

METHOD SUMMARY

- **main** – instantiate an object of this class. Make method calls to input and output.
- **input** – declare a Scanner object and read two integers from the keyboard – the radius of the base of the cone and the height of the cone.
- **surfaceArea** – *Arguments*: the radius of the base of the cone and the height of the cone. Calculate and return (as a double) the surface area of the cone. THIS SHOULD BE A STATIC METHOD.
- **volume** - *Arguments*: the radius of the base of the cone and the height of the cone. Calculate and return (as a double) the volume of the cone. THIS SHOULD BE A STATIC METHOD.
- **output** – display the results on the console screen. Make appropriate calls to `surfaceArea` and `volume`. Format all doubles to display no more than one decimal place. Do not display trailing zeros.

SAMPLE KEYBOARD INPUT
Enter the radius of the base: **8** *Enter the height of the cone:* **6**
SAMPLE OUTPUT
The surface area of a cone with a radius of 8 and a height of 6 is 452.4. **The volume of a cone with a radius of 8 and a height of 6 is 402.1.**

Ref: Lab 02 – ints, doubles, `DecimalFormat` class
Lab 02 – keyboard input
Lab 06 – `Math` methods.
Lab 12 – advanced methods.

Lab 23B - 70 points

OBJECTIVE

WAP that reads a sentence and a word from the keyboard. Insert the word into the sentence at the position that contains the first blank space. Make sure ONE space (and one space only) separates the inserted word from the word preceding it and the word following it.

FIELD SUMMARY

- **String sentence** – a sentence to be read from the keyboard
- **String word** – a word to be read from the keyboard.

METHOD SUMMARY

- **main** – instantiate an object of this class. Make method calls to `input`, `process`, and `output`.
- **input** – declare a Scanner object and read a sentence and a word from the keyboard.
- **process** – Locate the first blank space in the sentence and insert the word into that location using the substring method of the String class.
- **output** – display the altered sentence on the console screen.

SAMPLE KEYBOARD INPUT

Enter a sentence: **The brown fox jumped over the lazy dog's back.**
Enter a word: **quick**

SAMPLE OUTPUT

The quick brown fox jumped over the lazy dog's back.

Ref: Lab 14 – `String` methods

Lab 23C - 80 points

OBJECTIVE

WAP that reads a series of numbers from a data file ("numbers.dat") and stores them in an array of integers. Display the smallest number, the largest number, and the average of all the numbers. Format all doubles to display no more than two decimal places. Do not display trailing zeros.

FIELD SUMMARY

- **int[] list** – an array of integers
- **int count** – counter to keep track of the number of items in the array

METHOD SUMMARY

- **main** – instantiate an object of this class. Make method calls to `input`, `process`, and `output`.
- **input** – declare a Scanner object and read a series of integer values from a data file.
- **average** – *Arguments*: an array of integers and an integer representing the number of elements in the array. Calculate and return the average of all the numbers contained in the array. THIS SHOULD BE A STATIC METHOD.
- **output** – display the smallest and largest numbers in the array. Make a call to `average` to display the average of all the numbers in the array.

SAMPLE DATA FILE INPUT

88 2 14 26 321 44 89 76 44 57 13 28 66 95 854 32 65 58 99 4 13 8 78 46 72

SAMPLE OUTPUT

The smallest number is 2.
The largest number is 854.
The average of all the numbers is 91.68.

Ref: Lab 12 – advanced methods.
Lab 10 – arrays

Lab 23D - 90 points

OBJECTIVE

WAC called Cowboy (ADT) that stores information about famous western personalities.

FIELD SUMMARY

- **String firstName** – `this person's first name.`
- **String middleName** – `this person's middle name.`
- **String lastName** – `this person's last name.`
- **int born** – `the year this person was born.`
- **int died** – `the year this person died.`
- **String profession** – `this person's profession.`
- **String alias** – `this person's alias (the name that made him famous).`

METHOD SUMMARY

- **constructor** – *Arguments:* a value for every field in the class. Initialize all the fields in the class using the values received as arguments..
- **getFirstName** – Returns this person's first name.
- **setFirstName** – *Arguments:* A string containing this person's first name. Set the first name field to the name received as an argument.
- **getMiddleName** – Returns this person's middle name.
- setMiddleName – *Arguments:* A string containing this person's middle name. Set the middle name field to the name received as an argument.
- **getLastName** – Returns this person's last name.
- **setLastName** – *Arguments:* A string containing this person's last name. Set the last name field to the name received as an argument.
- **getYearBorn** – Returns this year this person was born.
- **setYearBorn** – *Arguments:* An integer containing the year this person was born. Set the born field to the value received as an argument.
- **getYearDied** – Returns this year this person died.
- **setYearDied** – *Arguments:* An integer containing the year this person died. Set the died field to the value received as an argument.
- **getProfession** – Returns this person's profession.
- **setProfession** – *Arguments:* A string containing this person's profession. Set the profession field to the profession received as an argument.
- **getAlias** – Returns this person's alias (if any).
- **setAlias** – *Arguments:* A string containing this person's alias. Set the alias field to the alias received as an argument.
- **equals** – *Arguments:* An object to be compared with this Cowboy object. Two Cowboy's are the same if their first and last names are the same.
- **toString** – Returns a string representation of this Cowboy object. The `toString` method should return the Cowboy's first, middle, and last names - IF the Cowboy has a middle name.

It should return only the first and last names if the Cowboy does not have a middle name. (If the middle name equals null then the Cowboy does not have a middle name). Following the name, the `toString` method should return the word AKA (also known as) followed by the Cowboy's alias (if he has an alias – if he does not have an alias then omit this part). Finally, the `toString` method should return a hyphen followed by the Cowboy's profession.

Henry McCarty AKA Billy The Kid - Outlaw
William Matthew Tilghman AKA Bill Tilghman - Lawman
Jesse Woodson James - Outlaw

Double click on CowboyTester.bat to test your `Cowboy` class. Output is shown below.

Henry McCarty AKA Billy The Kid - Outlaw
William Matthew Tilghman AKA Bill Tilghman - Lawman
Jesse Woodson James - Outlaw

Henry McCarty (1859 - 1881)
William Tilghman (1854 - 1924)
Jesse James (1847 - 1882)

Jesse James equals Jesse James: true
Jesse James equals Henry McCarty: false

Ref: Lab 18 – Abstract Data Types

Lab 23E - 100 points

OBJECTIVE

WAP that reads information pertaining to famous western personalities from a data file ("cowboys.dat") and stores them in an array of Cowboy objects. If the cowboy does not have a middle name or an alias then those fields should be null. Both fields contain a hyphen to indicate the field is not used.

FIELD SUMMARY

- **Cowboy[] cowboys** – an array of Cowboy objects.
- **int count** – a counter to keep track of the number of cowboys actually stored in the array of Cowboy objects.

METHOD SUMMARY

- **main** – instantiate an object of this class. Make method calls to `input`, and `output`.
- **input** – declare a `Scanner` object to read the cowboy data from a data file ("cowboys.dat"). Store the information read from the data file in the array `cowboys`. If the middle name field or the alias field contain a hyphen "-" then pass null as the argument for that field.
- **output** – for every Cowboy in the array of Cowboy's, display the cowboy's name, alias, and occupation using the Cowboy's `toString` method. In addition, display the year the Cowboy was born and the year he died inside a set of parentheses.

SAMPLE DATA FILE INPUT

```
Earp/Wyatt/Berry Stapp/1848/1929/Lawman/-
Hickok/James/Butler/1837/1876/Lawman/Wild Bill Hickok
Hardin/John/Wesley/1853/1895/Outlaw/-
Holiday/John/Henry/1851/1887/Gunfighter/Doc Holiday
Masterson/William/Barclay/1853/1921/Lawman/Bat Masterson
Parker/Robert/LeRoy/1866/1908/Outlaw/Butch Cassidy
Longabaugh/Harry/Alonzo/1867/1908/Outlaw/Sundance Kid
Parker/Isaac/Charles/1838/1896/Judge/Hanging Judge Parker
Garrett/Patrick/Floyd/1850/1908/Lawman/Pat Garrett
Bean/Phantly/Roy/1825/1903/Justice of the Peace/The Law West Of The Pecos
McCarty/Henry/-/1859/1881/Outlaw/Billy The Kid
```

SAMPLE OUTPUT

Wyatt Berry Stapp Earp - Lawman (Born: 1848 Died: 1929)
James Butler Hickok AKA Wild Bill Hickok - Lawman (Born: 1837 Died: 1876)
John Wesley Hardin - Outlaw (Born: 1853 Died: 1895)
John Henry Holiday AKA Doc Holiday - Gunfighter (Born: 1851 Died: 1887)
William Barclay Masterson AKA Bat Masterson - Lawman (Born: 1853 Died: 1921)

Robert LeRoy Parker AKA Butch Cassidy - Outlaw (Born: 1866 Died: 1908)
Harry Alonzo Longabaugh AKA Sundance Kid - Outlaw (Born: 1867 Died: 1908)
Isaac Charles Parker AKA Hanging Judge Parker - Judge (Born: 1838 Died: 1896)
Patrick Floyd Garrett AKA Pat Garrett - Lawman (Born: 1850 Died: 1908)
Phantly Roy Bean AKA The Law West Of The Pecos - Justice of the Peace (Born: 1825 Died: 1903)
Henry McCarty AKA Billy The Kid - Outlaw (Born: 1859 Died: 1881)

Ref: Lab 10 – arrays
Lab 18 – Abstract Data Types