

UNIT 3. Control structures in Oriented-Object Programming in Java

| | | |
|-------|---|----|
| 1 | Introduction..... | 4 |
| 2 | Control structures in Java | 4 |
| 2.1 | Introduction | 4 |
| 2.2 | Decision making or conditional statements..... | 6 |
| 2.2.1 | Nested if..... | 8 |
| | Guide Activity..... | 11 |
| | Independent Activity..... | 11 |
| 2.3 | Switch statement in Java..... | 12 |
| | Menus and switch Statements | 16 |
| 2.3.1 | Enum Types and Java Switch | 18 |
| 2.4 | Iterative statements in Java | 21 |
| 2.4.1 | Variables in loop statements. | 22 |
| | Counters. | 22 |
| | Accumulators. | 23 |
| | Maximums and Minimums. | 23 |
| 2.4.2 | The Basic While Loop | 24 |
| | Infinite loops | 26 |
| 2.4.3 | Loops for menu | 27 |
| 2.4.4 | Nested while loop | 30 |
| | Cornell notes. Activity..... | 33 |
| | Activity..... | 33 |
| 2.4.5 | The do .. while statement | 33 |
| 2.4.6 | For statement..... | 38 |
| 2.4.7 | Comparative among Java loop | 42 |
| 3 | Generic Classes and Interfaces in Java | 44 |
| 3.1 | Generic Interfaces..... | 48 |
| 4 | Functional interfaces and lambdas. Complex Java types. | 51 |
| 4.1 | Anonym classes | 52 |
| 4.2 | Anonym class from a class. Constructors..... | 52 |
| 4.2.1 | Anonym classes from interfaces. | 54 |
| 4.3 | Functional interfaces | 56 |
| 4.4 | Lambda expressions. | 61 |
| | Lambda single expression | 61 |
| | Lambda block expressions | 62 |
| | Practice | 64 |
| 4.5 | Predefine Functional Interfaces in Java..... | 65 |
| 4.5.1 | Function composition | 66 |
| 4.5.2 | Predicate Functional Interface..... | 66 |
| | Cornell notes:..... | 68 |
| | Predicate composition | 70 |
| | Cornell notes..... | 70 |
| | Practice | 70 |
| 4.5.3 | The Consumer Interface..... | 71 |
| | Practice | 73 |
| 4.5.4 | Supplier Interface | 74 |
| | Practice | 76 |
| 4.5.5 | The Function Functional Interface..... | 77 |

| | | |
|-------|--|-----|
| 4.5.6 | The Unary Operator Interface | 80 |
| | Activity..... | 81 |
| 4.5.7 | Bifunction functional interface | 82 |
| 4.5.8 | Primitive types in functional interfaces | 83 |
| 5 | Functional programming | 85 |
| | Categorization of Programming Paradigms | 85 |
| | Functional Programming | 86 |
| 5.1 | Java 8 and Functional programming | 88 |
| | Functional programming basics | 89 |
| 5.2 | Functions are first-class objects/ first-class members of the language | 89 |
| 5.3 | Pure functions. Stateless. | 93 |
| 5.3.1 | High-order functions..... | 95 |
| 5.3.2 | Stateless..... | 99 |
| 5.3.3 | No side effects..... | 100 |
| 5.3.4 | Immutable objects..... | 101 |
| 5.3.5 | Favor recursion over loops..... | 103 |
| 6 | Recursion..... | 103 |
| 6.1 | The Java Runtime Stack | 104 |
| 6.1.1 | ¿What is a stack? | 104 |
| 6.2 | The Java Runtime Stack | 104 |
| | The Garbage Collector | 106 |
| 6.3 | Recursion | 107 |
| 6.4 | Practice | 113 |

Color coding

Yellow: important definitions. / code statements of high relevance

Blue: concepts and topics / variable declarations

Red: key topics/ issues

1 Introduction

This unit is aimed to unravel the controls structures we use in java to convey the flow of our programs. Some of the control structures has been already introduced in Unit 2, such as the if .. else control statement. As a matter of a fact, Inheritance, properties and method definitions within a class can be considered as control structures, even if the most purist and classic developers would be in discord with this last statement.

Moreover, we will introduce the concept of Generic Classes and Interfaces. This kind of structures can receive the type used for some of its properties as class parameters. The idea behind generic classes is making classes work uniformly with different types.

Finally, we will introduce the Java 8 concept of Functional interface, and lambda expressions. It we will complete that we started in the previous unit, so that Functional Interfaces and lambda expressions can be approach as data types and parameters or arguments. It is an unconventional programming style that emanates from the functional programming, an style of programming that does not declare variables or do assignation. On the contrary, this programming style solve algorithms through functions definitions and function calls.

2 Control structures in Java

2.1 Introduction

As we motioned previously, in this section we will explain in detail the control flow structures we already introduced in the Unit 1 for Flowcharts. This time we are to provide training in these structures for Java. As a review, we should point out that there are four basic control flow structures in Java programming, which are the different language provision of statements:

1. Declaration statements.
 - a. Standard declaration
`int i;`
 - b. Inline declaration = declaration + assignation

```
SUVElectric myTeslaSUVElectric = new SUVElectric("red","Tesla",  
"Model S High Performace", 50000,30000,  
100, 17, 7);
```

2. Sequential statements

a. Asignation and expressions

i. Simple assignation

```
i=0;
```

ii. Complex assignation: It may include operators, and method calls

```
i= i + Car.numberOfCarsCreated();
```

3. Method calls

```
Car.numberOfCarsCreated();
```

4. or block statements: a group of sequential statements surrounded by {}. When we

```
{  
  
    i= i + Car.numberOfCarsCreated();  
  
    System.out.println("My new SUV:" + myTeslaSUVElectric.toString());  
  
    System.out.println("Total cars created: " + Car.numberOfCarsCreated());  
  
    myTeslaSUVElectric.repaint("White");  
  
}
```

They can be part of methods, or the subsequent statements, conditionals Iterative Statements. Its main characteristic is that any sentence in the block method executes in order. So, in the previous block the first sentence to execute should be a the assignation `= i= i + Car.numberOfCarsCreated();`

As this assignation ends, the second to be executed would be

```
System.out.println("My new SUV:" + myTeslaSUVElectric.toString());
```

And so on , until reach the end of the statement block.

5. Conditional or selection statements: the control the flow of our program relying on logical conditions.
6. Iterative Statements: they execute a statement or block statement depending on conditions set up by the programmer.
7. Class declaration, Interface declaration and methods declaration.

In this unit we will start working on selection statements and iterative statements.

2.2 Decision making or conditional statements

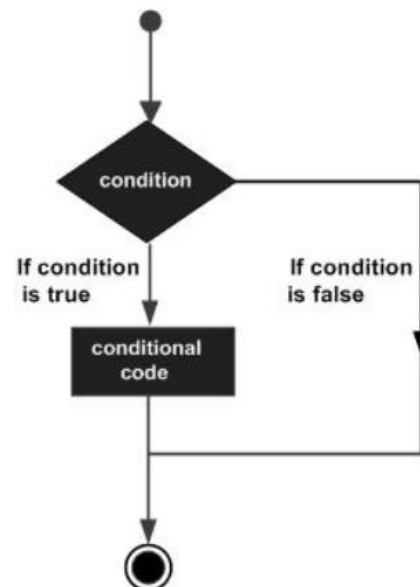
Conditional statements allow for conditions to be attached to a statement as to whether it will be executed. The most basic form is the if statement. In the code fragment below, the statement S is executed only if the boolean condition E evaluates to true. We have already study this in the previous unit.

For a simple statement if:

```
If (condition )  
    statement;
```

or for a block statement if:

```
if (condition) {  
    statement1;  
    statement2;  
    ...  
}
```



The statements included in the if statement are tagged as the If body.

A slight variation is the if-else statement which allows for an either-or choice. If the boolean condition evaluation is true and statement 1 and 2 are executed, then statement3 and 4 would not. On the contrary, if condition is false, statement 3 and 4 would be executed instead. Conditions are referred to also as test expression so that the program verified if the condition is satisfied any time they are executed.

Or

```
If (condition ) {  
    statement1;  
    statement2;  
..  
}  
  
else {  
  
    statement3;  
    statement4;  
..  
  
}
```

The third variation if else if

```
If (condition1) {  
    statement1;  
    statement2;  
}  
else if (condition2) {  
    statement3;  
    statement4;  
  
}
```

.... Numerous Else

```
else {  
  
    Statement n;  
    Statement n+1;  
  
}
```

We outlined the if sentence with and example in Unit2.

```
package ifstatements;  
  
public class IfSentence {  
  
    public static void main(String[] args) {
```

```

    int age=0;

    Scanner sc = new Scanner(System.in);

    System.out.println("Introduce the age of a Person");
    age = sc.nextInt();

    if (age>= 15 && age <= 18)
        System.out.println("The person is an teenager");
    else if (age> 18)
        System.out.println("The person is an adult");
    else if (age<18)

        System.out.println("The person is a minor age person");
    else if (age>=150)

        System.out.println("The person is not a human, maybe he is superman");

}

}

```

These are some of the possibilities that the if-statement provides. Anyway, in this subchapter we will try to extend the explanation of the if sentences adding more possibilities.

2.2.1 Nested if

Taking into consideration that the if sentence can contain a sentence block, you should realize that we can include an if sentence within a if sentence. In programming, we tagged this scenario as nested if. That results in if statement can be part of a block statement. The nested if should follow the next structure. You may use many variants of this structure. Multiple nested if, if else- if.

If (condition1) {

Statement1;

If (condition2) {

Statement2;
Statement3;

}


```

Java Programming
Ladera
    Statement4;
}

```

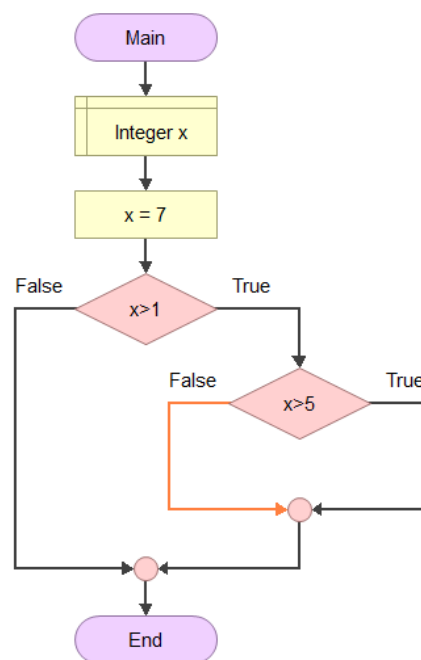
Carlos Cano

More or less the flowchart diagram look like and the related Java code as:

```

int x = 7;
if (x > 1) {
    if (x > 5) {
    }
}

```



To write clean code, it is highly recommended not to use very complex if-sentences. In addition, for long selections the Java switch statement is the most suitable choice. We will look over this statement in this section as well. Let us introduce a nested if example:

```

import java.util.Scanner;

public class NestedIf {

    public static void main(String[] args) {

        Integer number=0;
        int numDivisors=0;
        Scanner sc = new Scanner(System.in);

        System.out.println("Introduce an integer");
    }
}

```

```

        number = sc.nextInt();

        if (number%2==0) {

            System.out.println("the number " + number + " is divisible by 2");

            numDivisors++;
            if (number%3==0) {

                System.out.println("the number " + number + " is divisible by
3 and 6");
                numDivisors+=2;

            } else if (number%5==0){

                System.out.println("the number " + number + " is divisible by
5 and 10");
                numDivisors+=2;

            }

        }

    }

}

```

In this example, we have an outer if

```

if (number%2==0) {

```

and then, the inner if- else statement, enclosed in the outer if

```

    if (number%3==0) {

    } else if (number%5==0){

    }

}

```

Cornell notes:

We will do Cornell notes for this program commenting it line by line. But first, you may debug this program twice.

First execution: 12

Second execution :20

Debugging: Try to follow the program flow step by step. Verify variable values in the debug inspect area.

Commenting: After that, comment line by line what the program is doing:

Example:

```
//we read an Integer from the console
number = sc.nextInt();

//we check if the number is divisible by 2
if (number%2==0) {
```

Guide Activity

Write a Java program that reads a floating-point number and prints "zero" if the number is zero. Otherwise, print "positive" or "negative". Add "small" if the absolute value of the number is less than 1, or "large" if it exceeds 1,000,000.

Independent Activity

Write a Java program that take a number from the console. If the number is in the range of 1 and 9, print a salutation sentence, also if the number is divisible by two print “We are even” whereas if the number is divisible by 3, print “we like trios of objects”.

More difficult. Websearch. Find a way of generating random numbers in Java. Modify the former program to get a random number between 1 and 20 instead of your reading it from the console.

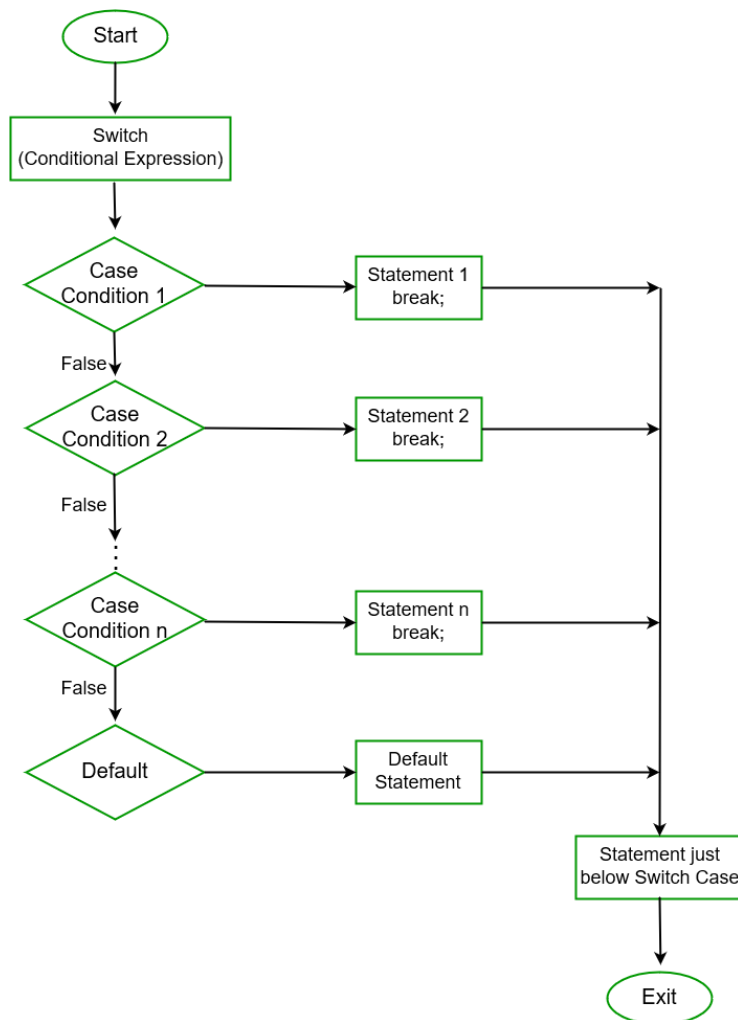
2.3 Switch statement in Java

Switch statements give us the power to choose one option among many alternatives. It allows you to test the value of an expression and, depending on that value, to jump directly to some location within the switch statement. Only expressions of certain types can be used. The value of the expression can be one of the primitive integer types int, short, or byte. In addition, the value expression can be a char type. Since Java 7, you can use the String typer for the value expression and the wrap classes for int (Integer), short (Short) and byte(Byte). The expression cannot be a decimal number, such as float or double. In modern Java versions, we can use **Enumerate Types** or switch expressions and cases

The positions that you can jump to are marked with case labels that take the form: "case constant:". This marks the position the computer jumps to when the expression evaluates to the given constant. As the final case in a switch statement, you can, optionally, use the label "default:", which provides a default jump point that is used when the value of the expression is not listed in any case label.

A switch statement, as it is most often used, has the form:

```
switch (expression i) {
    case constant-1:
        statements-1;
        break;
    case hconstant-2 i:
        statements-2;
        statements-3;
        statements4;
        break;
    .
    . // (more cases)
    .
    case constant-N i:
        statements-N;
        break;
    default: // optional default case
        statements-(N+1)
} // end of switch statement
```



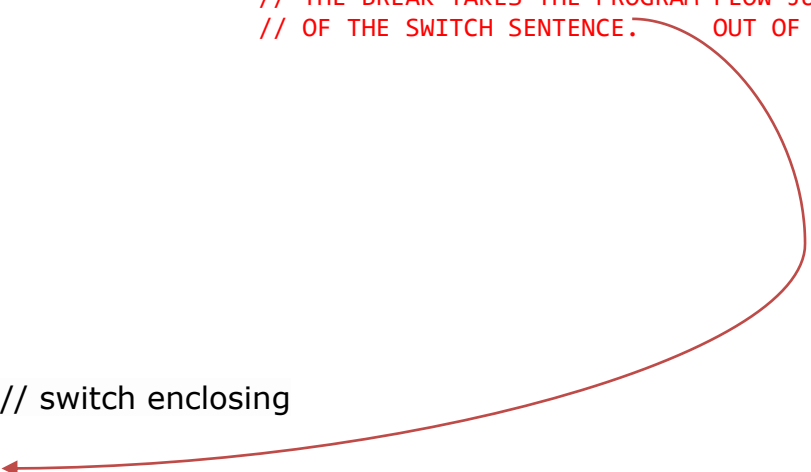
The break statements are technically optional. The effect of a break is to make the computer jump to the end of the switch statement. If you leave out the break statement, the computer will just forge ahead after completing one case and will execute the statements associated with the next case label. This is rarely what you want, but it is legal. (I will note here—although you won't understand it until you get to the next chapter—that inside a subroutine, the break statement is sometimes replaced by a return statement.)

in Java, **the break statement can be considered as a control flow structure per se**. Its function is taking the program flow out of the block statement where it is inserted. The break statement shows the same behavior in loops although it is considered a **bad practice to use the break statements in**. **Try to avoid it**

In the following example all breaks are inside the switch block

```
switch (number) { // (Assume N is an integer variable.)
    case 1:
        System.out.println("The number is 1.");
        break; // IF THE PROGRAM EXECUTE THIS BREAK

        // THE BREAK TAKES THE PROGRAM FLOW JUST TO THE END
        // OF THE SWITCH SENTENCE. OUT OF THE SWITCH BODY
    ...
    ...
    ...
} // switch enclosing
```



Let us have a look to the next example:

```
package switchstatement;

import java.util.Scanner;

public class NumericSwitch {

    public static void main(String[] args) {

        Integer number = 0;
        int numDivisors = 0;
        Scanner sc = new Scanner(System.in);

        System.out.println("Introduce an integer between 1 and 10");

        //we take an Integer from the console
        number = sc.nextInt();
    }
}
```

```
switch (number) { // (Assume N is an integer variable.)
case 1:
    System.out.println("The number is 1.");
    break;
case 2:
case 4:
case 8:
    System.out.println("The number is 2, 4, or 8.");
    System.out.println("(That's a power of 2!)");
    break;
case 3:
case 6:
case 9:
    System.out.println("The number is 3, 6, or 9.");
    System.out.println("(That's a multiple of 3!)");
    break;
case 5:
    System.out.println("The number is 5.");
    break;
default:
    System.out.println("The number is 7 or is outside the range 1 to
9.");
}

}
```

Activity.

1. We will add the “7” case for our switch. Execute the program, gather a 7 from the console and explain the results.

```
case 7:
    System.out.println("The number is 5.");
```

```
case 5:
    System.out.println("The number is 5.");
    break;
```

2. The next thing you will do is debugging the example twice. One for number=2 and the other for number=8. Are the results, the expected results? Jot it down your findings

It seems that the program it is not well written. Would you mind modifying the program and consider all the options from 1 to 9?

Menus and switch Statements

One application of switch statements is in processing menus. A menu is a list of options. The user selects one of the options. The computer has to respond to each possible choice in a different way. If the options are numbered 1, 2, . . . , then the number of the chosen option can be used in a switch statement to select the proper response.

In a console-based program, the menu can be presented as a numbered list of options, and the user can choose an option by typing in its number. Here is an example that could be used in a variation of the LengthConverter example from the previous section:
Iterative statements in Java.

The next example codifies a console app to convert a quantity from any measurement to inches. It gives the user for options. After selecting the option, it requires a quantity of units of the selected option to be converted to inches. Here the execution outcome:

What unit of measurement does your input use?

- 1. inches
- 2. feet
- 3. yards
- 4. miles

Enter the number of your choice:

4

Enter the number of miles:

63643634

The conversion to inches is: 4.03246065024E12

Steps to program this app:

1. To write the menu we use the println method

```
System.out.println("What unit of measurement does your input use?");
System.out.println();
System.out.println(" 1. inches");
System.out.println(" 2. feet");
System.out.println(" 3. yards");
System.out.println(" 4. miles");
System.out.println();
System.out.println("Enter the number of your choice: ");
```


2. We request the option from the console and we store it in the `optionNumber` variable:

```
optionNumber = sc.nextInt();
```

3. The switch option allows the program to distinguish among options and perform the selected conversion.
4. Each case will perform the specific conversion. We can have a look to case 2 as all cases operate similarly

case 2:

```
// 1. It reads the number of feet from the console
System.out.println("Enter the number of feet: ");
measurement = sc.nextDouble();
// 2. It converts from feet to inches by multiplying by 12
inches = measurement * 12;

//3. It offers the solution in the console output
System.out.println("The conversion to inches is: " + inches);
break;
```

The main advantage of the switch case is that it lets the program select among numerous possibilities. Once the option is selected, the only thing the program will need is codify the operation required by this option. In case 3, we needed to convert from feet to inches. To do so, we need to multiple feet by 12.

```
package switchstatement;

import java.util.Scanner;

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

        int optionNumber; // Option number from menu, selected by user.
        double measurement; // A numerical measurement, input by the user.
        // The unit of measurement depends on which
        // option the user has selected.
        double inches; // The same measurement, converted into inches.
        /* Display menu and get user's selected option number. */
```

```
Scanner sc = new Scanner(System.in);

System.out.println("What unit of measurement does your input use?");
System.out.println();
System.out.println(" 1. inches");
System.out.println(" 2. feet");
System.out.println(" 3. yards");
System.out.println(" 4. miles");
System.out.println();
System.out.println("Enter the number of your choice: ");
optionNumber = sc.nextInt();
/* Read user's measurement and convert to inches. */
switch (optionNumber) {
case 1:
    System.out.println("Enter the number of inches: ");
    measurement = sc.nextDouble();
    inches = measurement;
    System.out.println("The conversion to inches is: " + inches);
    break;
case 2:
    System.out.println("Enter the number of feet: ");
    measurement = sc.nextDouble();
    inches = measurement * 12;
    System.out.println("The conversion to inches is: " + inches);
    break;
case 3:
    System.out.println("Enter the number of yards: ");
    measurement = sc.nextDouble();
    inches = measurement * 36;
    System.out.println("The conversion to inches is: " + inches);
    break;
case 4:
    System.out.println("Enter the number of miles: ");
    measurement = sc.nextDouble();
    inches = measurement * 63360;
    System.out.println("The conversion to inches is: " + inches);

}

}
```

2.3.1 Enum Types and Java Switch

In this section, we are to introduce the use of Enumeration in switch statements. One of Java advantages regarding the switch statement is the capability of using enumerations as a switch expression.

Other new possibility that Java offered in its most recent versions (since Java 13) is the possibility of adding lists of elements to a case:

The property month is Month enumeration type:

```
private Month month;
```

We can use it as part of the switch expression:

```
switch (month)
```

We can also define a list for each case statement, `case DECEMBER, JANUARY, FEBRUARY:`.

```
switch (month)
{
    case DECEMBER, JANUARY, FEBRUARY:
        System.out.println("Cold, sometimes windy, others snowy.");
        break;
    case MARCH, APRIL, MAY:
        System.out.println("Spring is here, weather betters.");
        break;
    case JUNE, JULY, AUGUST:
```

```
package switchstatement;
```

```
// A Java program to demonstrate working on enum
// in switch case (Filename Test. Java)
import java.util.Scanner;

// An Enum class
enum Month
{
    JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE,
    JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER;
}

// Driver class that contains an object of "day" and
// main().
```

Ladera

```
public class ExampleEnumMonthOfTheYear {
```

```
    private Month month;
```

```
    // Constructor
```

```
    public ExampleEnumMonthOfTheYear(Month month)
    {
        this.month = month;
    }
```

```
    public Month getMonth() {
```

```
        return this.month;
```

```
    }
```

```
    // Prints a line about Day using switch
```

```
    public void daysLike()
    {
```

```
        switch (month)
        {
```

```
            case DECEMBER, JANUARY, FEBRUARY:
```

```
                System.out.println("Cold, sometimes windy, others snowy.");
                break;
```

```
            case MARCH, APRIL, MAY:
```

```
                System.out.println("Spring is here, weather betters.");
                break;
```

```
            case JUNE, JULY, AUGUST:
```

```
                System.out.println("Summer time, have a break and enjoy the weather, hot
and humid.");
```

```
                break;
```

```
            case SEPTEMBER, OCTOBER, NOVEMBER:
```

```
                System.out.println("Good weather, enjoy outside activities until november
take over.");
```

```
                break;
```

```
            default:
```

```
                System.out.println("Did you forget one month?");
                break;
```

```
        }
```

```
    }
```

```
    // Driver method
```

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

```
        String str = "JUNE";
```

```
        ExampleEnumMonthOfTheYear t1 = new
ExampleEnumMonthOfTheYear(Month.valueOf(str));
```

```
        System.out.println("How is it in " + t1.getMonth() + "?");
```

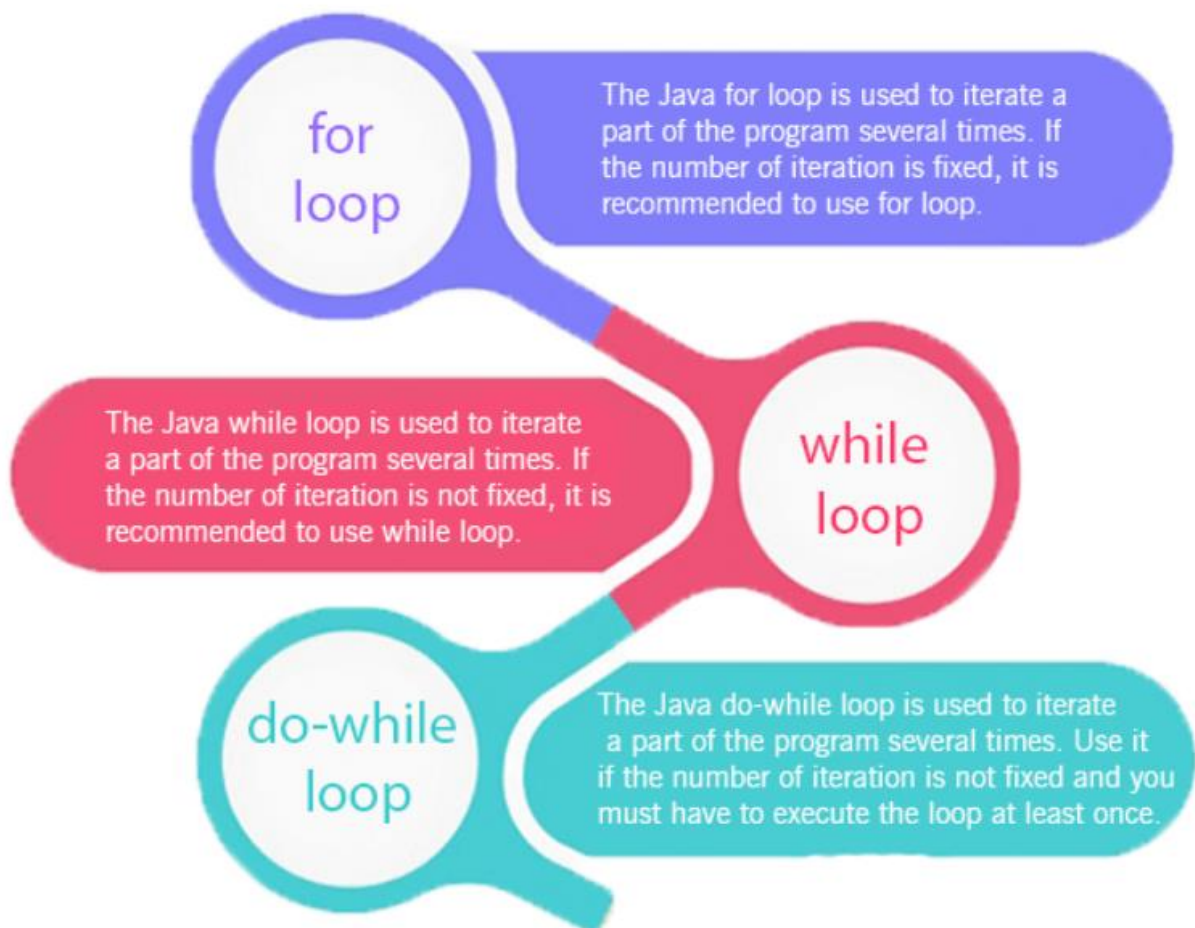
```
        t1.dayIsLike();  
    }  
}
```

2.4 Iterative statements in Java

The java programming language provides a set of iterative statements that are used to execute a statement or a block of statements repeatedly as long as the given test expression is satisfied, which means that is true.. The iterative statements are also known as looping statements or repetitive statements. Java provides the following iterative statements.

1. while statement
2. do-while statement
3. for statement
4. for-each statement

This control flow statements is one of the most powerful tools of programming languages. The ability to execute a block of sentences repeated times, allow programmers to implement more complex algorithm. From easy menu programs to difficult math operations such as factorial, prime numbers, are gracefully solve with the additions of loops.



2.4.1 Variables in loop statements.

They are objects used by a program and by the function they perform within it take a **special name**, modeling their operation due to their frequent use. These variables store data, such as the sum of a series of amounts, or the count of the number of students in class, or the maximum and minimum grade of a student, etc.

Counters.

A **counter** is an object that is used to count any event that can occur within a program. They usually count naturally from 0 to 1 in 1, although other account types required in some processes can be performed.

They are used by performing two basic operations on them:

1. **Initialization:** Any counter is initialized to 0 if you perform a natural account, or to another value if you want to perform another account type.

```
C1 = 0;  
CP = 2;
```

1. **Accounting or increment:** Each time the event to be counted is to increase the counter by 1 if a natural account is made or to another value if another account type is performed.

```
C1 = C1 + 1;  
CP = CP + 2;
```

Accumulators.

Accumulators. Accumulators are a special kind of variable that we basically use to update some data points across executors. One thing we really need to remember is that the operation by which the data point update happens **has to be an associated and commutative operation**.

These are objects that are used in a program to accumulate successive elements with the same operation. They are generally used to calculate sums and products, without discarding other possible types of accumulation.

Two basic operations will be performed on them for use:

- **Initialization:** Any accumulator is initialized to 0 if you make sums and 1 if you make products.

```
SUM = 0;  
PRODUCT = 1;
```

1. **Accumulation:** When the element to be accumulated by performing a reading or calculation is present in memory, the item is accumulated by means of the assignment:

```
SUM = SUM + QUANTITY;  
PRODUC = PRODUC * NUMBER;
```

Maximums and Minimums.

A maximum is a variable that saves the maximum value of a set of values. The maximum is initialized to a very small value so that we are a security that the first value stored in this variable is the largest.

A minimum is similar to the maximum only taking the smallest value of a series of values. The minimum is initialized to a maximum value.

2.4.2 The Basic While Loop

The block statement by itself really doesn't affect the flow of control in a program. The five remaining control structures do. They can be divided into two classes: loop statements and branching statements. You really just need one control structure from each category in order to have a completely general-purpose programming language. More than that is just convenience.

In this section, I'll introduce the while loop and the if statement. I'll give the full details of these statements and of the other three control structures in later sections.

A while loop is used to repeat a given statement over and over. Of course, it's not likely that you would want to keep repeating it forever. That would be an infinite loop, which is generally a bad thing, or a bad practice programming. (There is an old story about computer pioneer Grace Murray Hopper, who read instructions on a bottle of shampoo telling her to "lather, rinse, repeat." As the story goes, she claims that she tried to follow the directions, but she ran out of shampoo.

To be more specific, a while loop will repeat a statement over and over, but only so long as a specified condition remains true. A while loop has the form:

For one statement:

```
while(condition)
    Statement 1;
```

For a block statement:

```
while (condition) {

    statement 1;
    statement 2;
    statement 3;

    ....
    statement n;
}
```

Both the simple statement and the block statement are known as the while body.

Consider the next Java code. We have declare **the counter variable i**. The first time the program flow reaches the while statement, the condition `i<=10` is verified. Since the value

Ladera

of *i* is 1, the condition is true, the program flow goes into the loop block. In the block statement, it is written in the console the number of loop repetition, and then the value of *i* is increased one unit. As a result, "Loop repetition number: 1" is typed in the console, and the *i* variable value is 2.

This scheme is repeated until the while does not satisfy the conditions. In other words, to end the loop the value of *i* must be 11, so that 10 it is not greater than or equal to 10. In total, the while body is executed ten times. Remember, If and while conditions is known as test expressions

```
Loop repetition number: 1
Loop repetition number: 2
Loop repetition number: 3
Loop repetition number: 4
Loop repetition number: 5
Loop repetition number: 6
Loop repetition number: 7
Loop repetition number: 8
Loop repetition number: 9
Loop repetition number: 10
```

I recommend debugging this program step by step. Let's do it.

```
package loops;
```

```
public class SimpleWhile {
```

```
    public static void main(String[] args) {
        int i;

        i = 1;
        while (i <= 10) {
            System.out.println("Loop repetition number: " + i);
            i = i + 1;
        }
    }
}
```

Infinite loops

Provided that Humans we are, we can commit errors. Following that line of thoughts, it is not uncommon to code infinite loops. It is usually related to set up wrong conditions for the loop. Let me put it in other words, programs tend to be complex, and sometimes we make mistakes. For instance, in the next sample the condition is wrongly defined, and the program flow never gets out of the loop.

```
package loops;

public class InfiniteLoop {

    public static void main(String[] args) {
        int i;

        i = 1;
        while (i >= 0) {

            System.out.println("Loop repetition number: " + i);
            i = i + 1;
        }
    }
}
```

Pay attention to the loop test condition `while (i >= 0)`. The initial value of `i` is 1, thus the first time it reaches the while, `1>=0` is true. However, the loop block increases variable `i` by one in each repetition. Certainly, the condition `i>=0` will be satisfied. The programmer made a mistake setting the condition. Inevitably, the program will never leave the while statement. This code would be like the following example, given that the test expression (`i >= 0`) is always satisfied.

```
while (true) {

    System.out.println("Loop repetition number: " + i);
    i = i + 1;
}
```

In programming, we label this scenario **infinite loop. We need to avoid it at any cost.**

2.4.3 Loops for menu

In this section, we will combine the switch statement to create a menu, with a loop to keep the menu open until we press the Exit keyboard, 5.

Little modifications to the Switch Menu do we need to write. So as to create this new program, we will embed the code from the SwitchMenu.java example inside the body of a while. In the while test expression, we will control that the pressed key is not 5. On the contrary, provided that the input number from the console is 5, the program would exit the loop. Yellow highlighted the changes made with respect to the SwitchMenu.java example

Let me unveil the additions:

First, we check the optionNumber in the while test expression. Whether it is 5 the program flow exits the loop. On the contrary, the program execution continues to the while body. In the while body we have the **menu output**, **the input statement**, and the **switch statement** to perform the operation we have read from the console.

```
while (optionNumber != 5) {  
  
    System.out.println("What unit of measurement does your input use?");  
    System.out.println();  
    System.out.println(" 1. inches");  
    System.out.println(" 2. feet");  
    System.out.println(" 3. yards");  
    System.out.println(" 4. miles");  
    System.out.println(" 5. Exit the program");  
  
    System.out.println();  
    System.out.println("Enter the number of your choice: ");  
  
    optionNumber = sc.nextInt();  
  
    /* Read user's measurement and convert to inches. */  
    switch (optionNumber) {
```

Menu output

Console Input

Switch

The main advantage of this implementation is that we can repeat the menu infinite times until we press 5. This addition produces as a result that the user can perform conversion operations more than one time. The menu will be available to do measurement calculations until the user press 5.

We can verify this new functionality running the program. It is described what happen during the first while iteration. You should infer what happen in the second and third iteration. Explain it taking notes.

What unit of measurement does your input use?

1. inches
2. feet
3. yards
4. miles
5. Exit the program

Enter the number of your choice: **First while iteration**

1 **The user select 1**

Enter the number of inches:

3

The conversion to inches is: 3.0 **The program convert from inches to inches given that we selected switch case 2.**

What unit of measurement does your input use?

1. inches
2. feet
3. yards
4. miles
5. Exit the program

Enter the number of your choice: **Second while iteration**

4

Enter the number of miles:

2

The conversion to inches is: 126720.0

What unit of measurement does your input use?

1. inches
2. feet
3. yards
4. miles
5. Exit the program

Enter the number of your choice: **Last while iteration**

5

The program has reached the end. Bye!

WhileMenu.java

```
package loops;
```

```
import java.util.Scanner;
```

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

```
        int optionNumber = 1; // Option number from menu, selected by user.
```

```
double measurement; // A numerical measurement, input by the user.
// The unit of measurement depends on which
// option the user has selected.
double inches; // The same measurement, converted into inches.
/* Display menu and get user's selected option number. */

Scanner sc = new Scanner(System.in);

while (optionNumber != 5) {

    System.out.println("What unit of measurement does your input use?");
    System.out.println();
    System.out.println(" 1. inches");
    System.out.println(" 2. feet");
    System.out.println(" 3. yards");
    System.out.println(" 4. miles");
    System.out.println(" 5. Exit the program");
    System.out.println();
    System.out.println("Enter the number of your choice: ");

    optionNumber = sc.nextInt();

    /* Read user's measurement and convert to inches. */
    switch (optionNumber) {
    case 1:
        System.out.println("Enter the number of inches: ");
        measurement = sc.nextDouble();
        inches = measurement;
        System.out.println("The conversion to inches is: " + inches);
        break;
    case 2:
        System.out.println("Enter the number of feet: ");
        measurement = sc.nextDouble();
        inches = measurement * 12;
        System.out.println("The conversion to inches is: " + inches);
        break;
    case 3:
        System.out.println("Enter the number of yards: ");
        measurement = sc.nextDouble();
        inches = measurement * 36;
        System.out.println("The conversion to inches is: " + inches);
        break;
    case 4:
        System.out.println("Enter the number of miles: ");
        measurement = sc.nextDouble();
        inches = measurement * 63360;
        System.out.println("The conversion to inches is: " + inches);
        break;
    case 5:
        System.out.println("The program has reached the end. Bye! ");
        break;
    }
}
```

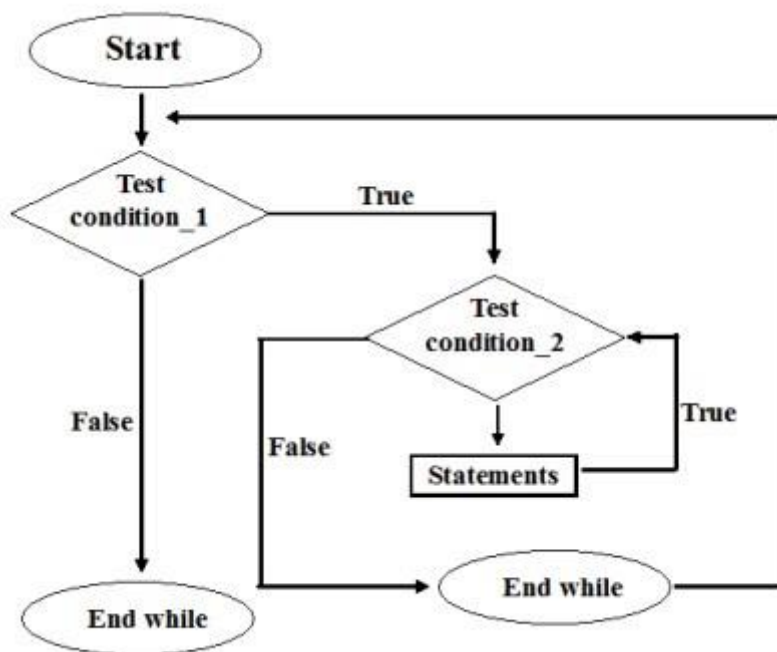
2.4.4 Nested while loop

Again, as previously we introduced for if statements, we can include while statements inside a while body. In other words, a loop within a loop. When a while loop exists inside the body of another while loop, it is known as nested while loop in Java. Initially, the outer loop executes once, and afterwards inner loop begins to execute. Execution of the inner loop continues until the condition of the inner loop is satisfied (until the test expression is false).

Once the Condition of the inner loop is satisfied, the flow of control comes out to the outer loop for next iteration. To improve our program performance, we have to control nested while, so as to not create inefficient whiles. You need to take into consideration that any time the outer loop iterates means that the inner loop has completed all his iterations. In this case, provided that the inner loop execute 100 iterations and the outer loop performs 60 iterations, we have a total of 60×100 iterations, six thousand.

To improve the performance and efficiency of our programs we try to reduce the number of iterations as much as we can. To achieve so, we try to define the optimal test expressions for our loops. Having say that, we can continue with an example of nested whiles.

The flow chart for nested whiles:



In the next example, we are created a program that calculate the forth power for each number of the outer while. So as to complete the calculation, we need an inner loop that repeats four times. Each time we multiple the external loop variable.

At a first glance, you could identify to counter variables, variable *i* for the outer while and variable *j* for the inner while. Moreover, the result variable, which is an accumulator, is initialized with a 1, `int result = 1;`. It serves for the purpose of calculating the fourth power for each value of the counter variable *i*. In any new outer loop iteration, we reset result to 1 with the aim of proceeding to a new power calculation.

NestedWhile.java

```

package loops;

public class NestedWhile {
    public static void main(String args[]) {
        int i = 1;
        int result = 1;

        while (i <= 3) {
            System.out.println("\n" + i + " " + "outer loop executed only once");
            System.out.println("input number to calculate the fourth power:" + i
+ " \n");

            int j = 1;
            result = 1;

```

```
        while (j <= 4) {
            result *= i;
            System.out.println(j + " " + "inner loop executed until to
completion");
            j++;
        }
        System.out.println("\nthe fourth power of the number " + i + " is "
+ result);
        i++;
    }
}
```

We can execute the program and analyze the results.

In the first iteration of the outer while, it only completes one iteration whereas the inner loop completes all of its iteration, four. The number to calculate the power to four is the value stored in the *i* variable, 1. The power operation outcome is 1.

1 outer loop executed only once

input number to calculate the fourth power:1 **i=1**

1 inner loop executed until to completion **j=1**

2 inner loop executed until to completion **j=2**

3 inner loop executed until to completion **j=3**

4 inner loop executed until to completion **j=4**

the fourth power of the number 1 is 1 **result=1 (1⁴=1)**

Subsequently, in the second iteration, the value of *i* is incremented by 1; therefore *i*=2.

2 outer loop executed only once

input number to calculate the fourth power:2 **i=2**

1 inner loop executed until to completion **j=1**

2 inner loop executed until to completion **j=2**

3 inner loop executed until to completion **j=3**

4 inner loop executed until to completion **j=4**

the fourth power of the number 2 is 16

Cornell notes. Activity

Following the previous executions, could you point out variable stored values for outer while iteration 3. Explain with your own words the program flow in these two iterations:

```
3 outer loop executed only once
input number to calculate the fourth power:3
```

```
1 inner loop executed until to completion
2 inner loop executed until to completion
3 inner loop executed until to completion
4 inner loop executed until to completion
```

```
the fourth power of the number 3 is 81
```

Activity.

Modify the program to control the number of iterations of the outer while. You should take the number of iterations from the console and store it in a variable `numberOfOuterIter`.

If this number is 5, the outer loop would perform five iterations.

Modify the program to read from the console the power you want to calculate. You can store this information in a variable labeled `powerTh`. For example, if the number is six the program is Compel to calculate the Sixth power.

2.4.5 The do .. while statement

Sometimes it is more convenient to test the continuation condition at the end of a loop, instead of at the beginning, as is done in the while loop. The `do..while` statement is very similar to the while statement, except that the word “while,” along with the condition that it tests, has been moved to the end. The word “do” is added to mark the beginning of the loop. A `do..while` statement has the form for a simple body statement.

```
do
    statement i
```

Ladera

```
while ( boolean-expression i );
```

For a block statement:

```
Do {  
    statement i;  
    statement i+1;  
    statement i +2;  
  
} while ( boolean-expression i );
```

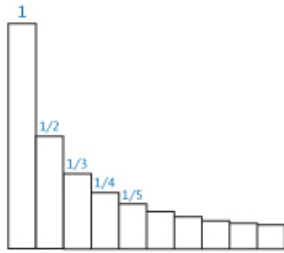
Note the semicolon, ';', at the very end. This semicolon is part of the statement, just as the semicolon at the end of an assignment statement or declaration is part of the statement. Omitting it is a syntax error. (More generally, every statement in Java ends either with a semicolon or a right brace, '}.')

To execute a do loop, the computer first executes the body of the loop—that is, the statement or statements inside the loop—and then it evaluates the boolean expression. If the value of the expression is true, the computer returns to the beginning of the do loop and repeats the process; if the value is false, it ends the loop and continues with the next part of the program. Since the condition is not tested until the end of the loop, the body of a do loop is always executed at least once.

As has been shown, the main difference between do-while and the while statement is the test expression location. As long as the program execution reaches the do while loop, it will execute the while body at least once owing that the test expression is evaluated at the end. On the contrary, in the while statement, the condition is evaluated at the beginning, thus in this case the condition is not satisfied the while body will not execute.

We are to set an example for the do- while statement. We will try to solve the algorithm problem of harmonic numbers.

HarmonicNumber () uses the same paradigm to evaluate the finite sum $H_n = 1 + 1/2 + 1/3 + \dots + 1/n$. These numbers, which are known as the harmonic numbers, arise frequently in discrete mathematics. Harmonic numbers are the discrete analog of the logarithm. They also approximate the area under the curve $y = 1/x$. You can use this example as a model for computing the values of other finite sums. A harmonic series (also overtone series) is the sequence of frequencies, musical tones, or pure tones in which each frequency is an integer multiple of a fundamental.



n=1 n=2 n=3 n=4 n=5

The harmonic series: 1, $\frac{3}{2}$, $\frac{11}{6}$, $\frac{25}{12}$, $\frac{137}{60}$

The objective of our program is to display each member of the harmonic series from 1 to n (we will read n from the consoles) and calculate the addition. We will try in this example to plan our program in advance and take notes. **We do Cornell notes commenting the program:**

What kind of variable do we need?

Counter -> to perform the while for the series element 1 to the element n
Accumulator -> to accumulate the addition of each member of the series
seriesNumber? To keep the current member
n, the series size. We need to obtain this information from the console

Identify all variables in the example with comments

What kind of test expression we will define?

We need to obtain the series elements from i to n. Consequently, we need to do n iterations. We might define the condition as:

i=1

Do {

} While (i<=n)

What are the goals of our program?

Display each element of the harmonic series.
Calculate the addition of all of them

Display the addition at the end of the loop.

Identify all the goals in the program

Let us define each member of the series

The harmonic series: $1, \overset{n=1}{3/2}, \overset{n=2}{11/6}, \overset{n=3}{25/12}, \overset{n=4}{137/60}, \overset{n=5}{\dots}$

Addition

For N=1 is 1

For N=2 is $1 + 3/2 = 5/2$

For N=3 is $1 + 3/2 + 11/6 = 26/6 = 13/3$

For N=4 is $1 + 3/2 + 11/6 + 25/12 = 77/12$

And so on:

DoWhileHarmonics.java

```
package loops;

import java.util.Scanner;

public class DoWhileHarmonics {

    public static void main(String[] args) {
        int i;

        i = 1;
```

```

//n, the series size. We need to obtain this
//information from the console

int n=1;
double seriesElement=1;
double seriesAddition=seriesElement;
Scanner sc = new Scanner(System.in);

System.out.println("How many elements of the harmonic series
do you want to calculate? "
                    + "\nType an integer number greater than or equal
to 1");

//n, the series size. We need to obtain this
//information from the console

n = sc.nextInt();

do {
    System.out.println("The " + i + " series number is "+
seriesElement);
    i++;
    seriesElement = seriesElement + (1/(double) i);
    seriesAddition=seriesAddition+ seriesElement;

} while (i<=n);

System.out.println("\nThe summatory of the " + n + "
elements of the Harmonic Series is "+ seriesAddition);

}

```

Here the execution of the program:

```

How many elements of the harmonic series do you want to calculate?
Type an integer number greater than or equal to 1
10
The 1 series number is 1.0
The 2 series number is 1.5
The 3 series number is 1.8333333333333333
The 4 series number is 2.0833333333333333
The 5 series number is 2.2833333333333333
The 6 series number is 2.4499999999999997
The 7 series number is 2.5928571428571425
The 8 series number is 2.7178571428571425
The 9 series number is 2.8289682539682537
The 10 series number is 2.9289682539682538

```

The summatory of the 10 elements of the Harm0nic Series is 25.23852813852813

2.4.6 For statement.

We turn in this section to another type of loop, the “for” statement. Any for loop is equivalent to some while loop, so the language doesn’t get any additional power by having for statements. But for a certain type of problem, a for loop can be easier to construct and easier to read than the corresponding while loop. It’s quite possible that in real programs, for loops actually outnumber while loops.

Structure of the form loop

```
for(initialization;condition;incr/decr)
    statement 1;
```

Or

```
for(initialization;condition;incr/decr) {
    statement 1;
    statement 2;

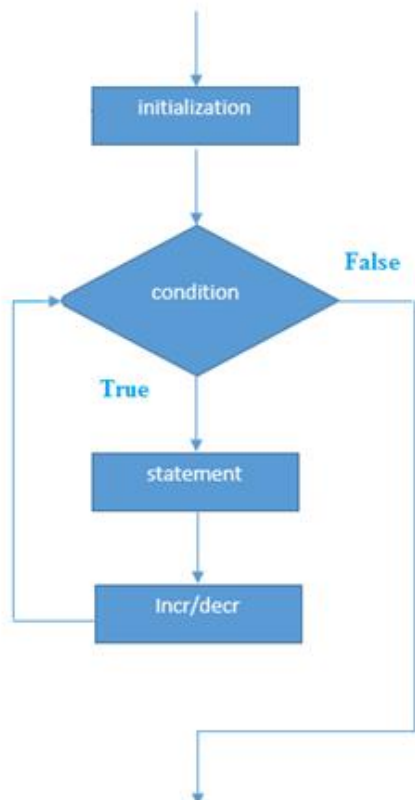
    ....
    Statement n;

}
```

Where:

1. **Initialization:** It is the initial condition which is executed once when the loop starts. Here, we can initialize the variable, or we can use an already initialized variable. It is an optional condition.
2. **Condition:** It is the second condition which is executed each time to test the condition of the loop. It continues execution until the condition is false. It must return boolean value either true or false. It is an optional condition.

3. **Statement:** The statement of the loop is executed each time until the second condition is false.
4. **Increment/Decrement:** It increments or decrements the variable value. It is an optional condition.



ForExample.java

```
package loops;

//Java Program to demonstrate the example of for loop
//which prints table of 1
public class ForExample {
    public static void main(String[] args) {
        // Code of Java for loop
        for (int i = 1; i <= 10; i++) {
            System.out.println(i);
        }
    }
}
```

Initialization: `int i = 1;`

Condition; `i <= 10`

Incr/Decr: `i++`

The execution order for this for would be:

First Iteration:

1. The initialization of the loop counter. `int i = 1;` It only executes once, as the program try to executes the for.
2. The condition `i <= 10` is satisfied. `1<=10`
3. The for body is executed `System.out.println(i);`
4. The increment/decrement expressions `i++` `i=2`

Second Iteration

1. The condition `i <= 10` is satisfied. `2<=10`
2. The for body is executed `System.out.println(i);`
3. The increment/decrement expressions `i++` `i=3`

....

Tenth iteration

1. The condition `i <= 10` is not satisfied. `11<=10`
2. The program flow leaves the for loop

Practice.

Create a class ForProductTable.

Using a nested for try to display the product table for the first ten numbers. You can use this structure for the nested for:

```
for (int i = 1; i <= 10; i++) {  
    for (int j = 1; j <= 10; j++) {  
          
    }  
}
```

The program executions should look alike the following:

Product table for 1

1X1=1
1X2=2
1X3=3
1X4=4
1X5=5
1X6=6
1X7=7
1X8=8
1X9=9
1X10=10

Product table for 2

2X1=2
2X2=4
2X3=6
2X4=8
2X5=10
2X6=12
2X7=14
2X8=16
2X9=18
2X10=20

... .

... .

Product table for 10

10X1=10
10X2=20
10X3=30
10X4=40

Ladera

10X5=50

10X6=60

10X7=70

10X8=80

10X9=90

10X10=100

2.4.7 Comparative among Java loop

Here a comparative table for Java loops statements.

| Comparison | for loop | while loop | do while loop |
|--------------|--|--|---|
| Introduction | The Java for loop is a control flow statement that iterates a part of the programs multiple times. | The Java while loop is a control flow statement that executes a part of the programs repeatedly on the basis of given boolean condition. | The Java do while loop is a control flow statement that executes a part of the programs at least once and the further execution depends upon the given boolean condition. |
| When to use | If the number of iteration is fixed, it is recommended to use for loop. | If the number of iteration is not fixed, it is recommended to use while loop. | If the number of iteration is not fixed and you must have to execute the loop at least |

| | | | |
|---|---|--|--|
| | | | once, it is recommended to use the do-while loop. |
| Syntax | <pre>for(init;condition;incr /decr){ // code to be executed }</pre> | <pre>while(condition){ //code to be executed }</pre> | <pre>do{ //code to be executed }while(condition) ;</pre> |
| Example | <pre>//for loop for(int i=1;i<=10;i++){ System.out.println(i); }</pre> | <pre>//while loop int i=1; while(i<=10){ System.out.printl n(i); i++; }</pre> | <pre>//do-while loop int i=1; do{ System.out.printl n(i); i++; }while(i<=10);</pre> |
| Syntax for infinite loop | <pre>for(;;){ //code to be executed }</pre> | <pre>while(true){ //code to be executed }</pre> | <pre>do{ //code to be executed }while(true);</pre> |

3 Generic Classes and Interfaces in Java

At its core, the term **generics** means **parameterized types**. Parameterized types are important because they allow you to create classes, interfaces, and methods in which the type of data *operates on* is specified as a parameter. A class, interface, or method that works with a parameter type is called **generic**, such as a generic class or method.

The main advantage of **generic code** is that it will automatically work with the data type passed to its type of parameter. Many algorithms are logically the same, regardless of the type of data to which they apply. For example, a Quicksort (sorting algorithm) is the same if you are sorting elements of type Integer, String, Object, or Thread. **With generics, you can define an algorithm once, regardless of any specific type** of data, and then apply that algorithm to a wide variety of data types without any additional effort.

From Java SE 5, we can create classes whose type is indicated at compile time. Look at the following example. We define a generic class that receives a type T. **class Generic<T>**. The operator <> is known as a **Diamond operator**. They allow us to create classes that handle different types.

If you look at the example, we have created two objects of our generic class. One for Double, another for Integer. It allows us to define the type that our generic class will use from compile time. Let us break down the next class:

We have declared a generic class of generic T type. The class has two properties Type T, operand1 and operand2.

```
public class GenericClassExample<T>

private T operand1;
private T operand2;
```

We perform some operations through methods for these generic operands, no matter which is the type. In the first of them the Type class is returned as a String, whereas in the second one both operands are compared.

```
public String displayType () {

    return operand1.getClass().toString();
```

```
public boolean comparation() {  
    return operand1.equals(operand2);  
}
```

In the third one, we compare operand1 with a value pass as a parameter.

```
public boolean comparationExternal(T op3) {  
    return operand1.equals(op3);  
}
```

To create objects for this class, we need to pass the type as a parameter using the diamond operator. In the first object, we pass Double as a parameter. Therefore, in the constructor call we need to pass Double typed values as a parameter.

```
GenericClassExample<Double> genericDouble = new GenericClassExample<Double>  
(Double.valueOf(1), Double.valueOf(2));
```

Moreover, for each method in the class that receive a T parameter, since we have defined the parametrized T as Double, we must pass a Double as a parameter. For example, you can have a glance at the comparationExternal method. It receives a parameter op3 typed T.

```
public boolean comparationExternal(T op3) {  
    return operand1.equals(op3);  
}
```

When we call this method from the object genericDouble, which T is Double in the object creation, we pass as a parameter a Double `Double param =1.0;`.

```
System.out.println(" operand1 are the parameter? The response is " +  
genericDouble.comparationExternal(param));
```

We could do the same for getters and setters. Try to introduce this line to your code and verify whether that works.

```
genericDouble.setOperand2(param);
```

GenericClassExample.java

```
package generic;

public class GenericClassExample<T> {

    private T operand1;
    private T operand2;

    public GenericClassExample(T operand1, T operand2) {

        this.operand1=operand1;
        this.operand2=operand2;

    }

    public T getOperand1() {
        return operand1;
    }

    public void setOperand1(T operand1) {
        this.operand1 = operand1;
    }

    public T getOperand2() {
        return operand2;
    }
}
```

```
public void setOperand2(T operand2) {
    this.operand2 = operand2;
}

public String displayType () {
    return operand1.getClass().toString();
}

public boolean comparation() {
    return operand1.equals(operand2);
}

public boolean comparationExternal(T op3) {
    return operand1.equals(op3);
}

@Override
public String toString() {
    return "GenericClassExample [operand1=" + operand1 + ", operand2=" +
operand2 + "]\n";
}

public static void main(String[] args) {

    Double param =1.0;

    GenericClassExample<Double> genericDouble = new GenericClassExample<Double>
(Double.valueOf(1),Double.valueOf(2));

    System.out.println("My generic class is typed as " +
genericDouble.displayType());

    System.out.println("Both operands are equal? The response is " +
genericDouble.comparation());

    System.out.println(" operand1 are the parameter? The response is " +
genericDouble.comparationExternal(param));

    GenericClassExample<Integer> genericInteger = new GenericClassExample<Integer>
(Integer.valueOf(1),Integer.valueOf(2));
```

```

        System.out.println("My generic class is typed as " +
genericInteger.displayType());

        System.out.println("Both operands are equal? The response is " +
genericInteger.comparation());

    }
}

```

We can extract several conclusions from this example. The main one might be that generic classes allow to create classes that manage multiple types, which is a powerful tool. This technique is intended for wrap classes and collection classes. Collection classes are classes that manage a collection of objects we will study in this course.

Generic classes allow us to restrain the T type. We can make the T type extends a class or implements an Interface. Let me illustrate the last statement. To do so, you can change the class declaration in the former example.

```
public class GenericClassExample<T extends Number>
```

Number is the parent class of wrapper numeric classes in Java, such as Integer, Double, Long and so on. Providing T extends from Number, we are forcing T to be of Number Type or one of its subclasses, Double, Long, etc. This is a mean provided by Java to restrict the T Type to a more concrete scope, in this cases Java numeric Types. Taking into consideration **this declaration, we cannot create a typed String object of this class, such as new GenericClassExample<String>.**

Practice

Try to create and equals method for the preceding generic class. The equals method would return true if operand1, and operand 2 are equal for both objects.

3.1 Generic Interfaces

Generic interfaces follow the same principles of Generic classes. However, they are even more useful in Java so that we only declared abstract methods in Java, and we can implement when we know the T type we have passed as a parameter to create objects of

Let us address GenericInterfaces programming an example. We have define the interface GenericInterfaceExample<T> with generic Type T

```
package generic;

public interface GenericInterfaceExample<T> {

    T addition(T op1, T op2);
    T subtraction(T op1, T op2);
    T product(T op1, T op2);
    T division(T op1, T op2);

}
```

We can do multiple implementations of this interface with different types. To illustrate this, consider the next two classes:

The ClassDouble class implements the interface passing as a parameter Double for T. Now, the method implementations of the interface receive Double as parameters. We can do a concrete implementation of this methods based on a specific type Double. The response for division(Double.valueOf(3), Double.valueOf(2)) should be 1.5.

```
package generic;

public class ClassDouble implements GenericInterfaceExample<Double>{

    @Override
    public Double addition(Double op1, Double op2) {
        // TODO Auto-generated method stub
        return op1+op2;
    }

    @Override
    public Double subtraction(Double op1, Double op2) {
        // TODO Auto-generated method stub
        return op1-op2;
    }

    @Override
    public Double product(Double op1, Double op2) {
```

Ladera

```

        // TODO Auto-generated method stub
        return op1*op2;
    }

    @Override
    public Double division(Double op1, Double op2) {
        // TODO Auto-generated method stub
        return op1/op2;
    }
}

```

Now have a look to the following implementation. It is similar but this time T is typed Long. The result of calling the method `division(Long.valueOf(3), Long.valueOf(2))` would be 1, since our implementation is based on an integer type.

```

package generic;

public class ClassLong implements GenericInterfaceExample<Long> {

    @Override
    public Long addition(Long op1, Long op2) {
        // TODO Auto-generated method stub
        return op1+op2;
    }

    @Override
    public Long substraction(Long op1, Long op2) {
        // TODO Auto-generated method stub
        return op1-op2;
    }

    @Override
    public Long product(Long op1, Long op2) {
        // TODO Auto-generated method stub
        return op1*op2;
    }

    @Override
    public Long division(Long op1, Long op2) {
        // TODO Auto-generated method stub
        return op1/op2;
    }
}

```

All in all, the generic types lead our code to simulate similar behaviors for similar types, although they keep their own characteristics. The division for integer types is slightly different that the division for Doubles.

4 Functional interfaces and lambdas. Complex Java types.

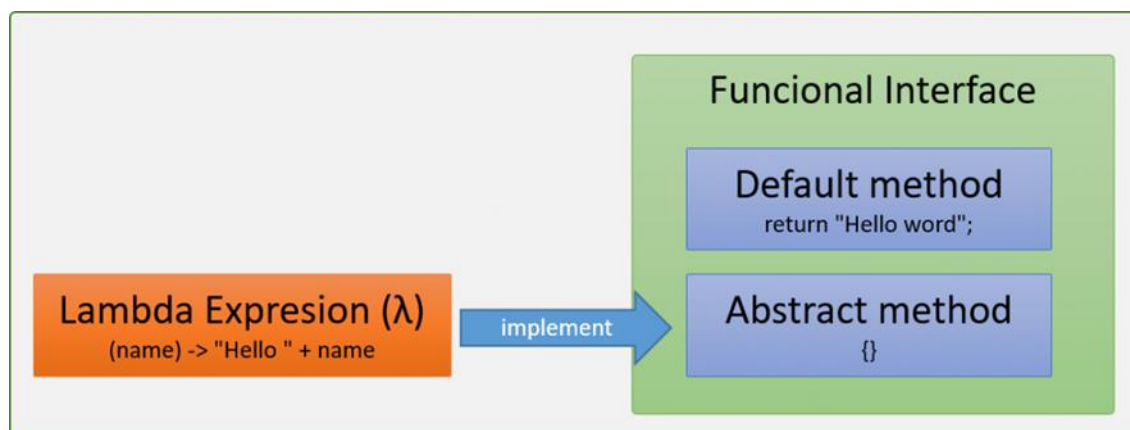
We remember from our knowledge in language theory **three paradigms of programming languages: imperative, functional, and declarative**. Java has traditionally been an imperative or declarative programming language, based on mutability, assigning values to variables, and transporting or transforming data from one variable or data structure to another.

The functional paradigm is based on the mathematical concept of **function**. Programs written in **functional languages** will consist of a set of function definitions (understanding these not as classic subprograms of an imperative language) along with the **arguments** on which they apply.

Starting with java version 8 a new programming paradigm is available in Java. That is **functional programming**. Perform code using single or basically functions and lists as if we were using a functional language like Lisp. This new way of programming is based on:

- **Functional interfaces:** interfaces that offer **only an abstract method**. I mean, they do one job.
- **Lambda expressions:** they are based on the **lambda calculation**. The concept is the declaration of anonymous **functions**. In lambda calculation a function can be declared anonymously. **For example, $\text{Square}(x)x^2$** can be defined as $x.x^2$ or $(x) \rightarrow x^2$. This way we will **define our functions in java**. And we'll solve our algorithms using basically functions.

The goal is to **overwrite that abstract method provided by the functional interface** with a **Lambda expression**. If the interface is not functional, a lambda expression cannot be passed to it. We save instructions and lines of code by doing it this way. You can say lambda expression represents an anonymous function that is contributed to the functional interface that defines the anonymous function. They bring real action to the definition



4.1 Anonym classes

An **anonymous inner class** is an internal class shape that is **declared and instantiated with a single declaration**. As a result, there is **no name for the class that can be used elsewhere** in the program; I mean, it's anonymous.

Anonymous classes are **typically used in situations** where you need to be able to create a **lightweight class** that is passed as a **parameter or argument**. This is usually done with an interface.

For example, in the next subchapter, we will create an anonym class from another class and from an interface. This practice is widely extended in Java nowadays.

4.2 Anonym class from a class. Constructors

The idea is achieving inheritance avoiding the need of adding a file .java and write a new class. Sometimes, we need a new class that slightly modify the parent class. In this scenario, we do not need to write the code for a new class, we can create the class in runtime as it will be shown in the next example. We create anonym classes by means of overriding some of parent class methods in runtime.

In the following program, you can find a class named as BaseClassForAnnonym. It contains a constructor and the method methodName. What it is attempted in the program is creating a new object of a new anonymous class. So as to obtain this we follow these steps.

1. We create a variable parent class type, BaseClassForAnonym.
`BaseClassForAnonym anon =`
2. We use new to invoke the BaseClassForAnonym constructor
`new BaseClassForAnonym("Catherine")`
3. At this point, Java let programmers dynamically override the parent class method methodName, adding a block after the constructor call.

```
{
    @Override
    public void methodName(String name) {
        System.out.println("Overriding the method name anonymously:" + name);
    }
};
```

Considering the BaseCallsForAnonym It turns out that the new object created is different from one created from the original class. The reason is that the methodName are different. As a result, we have an object new class, but this new class is unnamed, an anonym class.

The method for an object of the class BaseCallsForAnonym is

```
public void methodName(String name) {
    System.out.println("The name is anonymously:" + name);
}
```

The new object created has this methodName method:

```
@Override
public void methodName(String name) {
    System.out.println("Overriding the method name anonymously:" + name);
}
```

As you can verify, they are different. Anonym classes are intended to create classes from interfaces as we will explain in the next example.

```
BaseClassForAnonym anon = new BaseClassForAnonym("Catherine") {
    @Override
    public void methodName(String name) {
        System.out.println("Overriding the method name anonymously:" + name);
    }
};
```

BaseClassForAnonym.java

```
public class BaseClassForAnonym {
    public BaseClassForAnonym (String nombre) {

    }

    public void methodName(String name) {
```

```
        System.out.println("The name is anonymously:" + name);
    }

    public static void main(String[] args) {

        BaseClassForAnonym anon = new BaseClassForAnonym("Catherine") {
            @Override
            public void methodName(String name) {

                System.out.println("Overriding the method name anonymously:" + name);
            }
        };

        anon.methodName("Mildred");
    }
}
```

4.2.1 Anonym classes from interfaces.

This case is the most usual in Java programming. The Java API framework latest additions, Android SDK framework (for android developers), are designed to create anonymous classes from interfaces. The procedure is similar to the preceding example:

- a. An interface InterfaceAnonym has been define within this example.

```
interface InterfaceAnonym {

    public void methodName(String name);

}
```

- b. We declare an interface variable. We have already introduced in unit 2 that we can create variables Interface type for classes that implement this interface.

```
InterfaceAnonym anon =
```

- c. Java allows calls to an unexisting interface constructors (Interfaces do not have constructor) if we override the abstract methods of this interface

- d. In runtime, the abstract method `methodName` is implemented by the anonym class.

```
@Override
    public void methodName(String name) {
        System.out.println("Overriding the method name anonymously:" +
name);
    }
};
```

- e. Again, the outcome is an object from new class created from the interface. But this class does not exist in any file. Moreover, it has no given name, thus it is anonym.

```
package introductionfunctionalprogrammin;

public class AnonymClassFromInterfaceExample {
    interface InterfaceAnonym {
        public void methodName(String name);
    }

    public static void main(String[] args) {
        InterfaceAnonym anon = new InterfaceAnonym() {
            @Override
            public void methodName(String name) {
                System.out.println("Overriding the method name anonymously:" +
name);
            }
        };

        anon.methodName("Mildred");
    }
}
```

Anonym classes and objects are a key concept to develop lambda expressions in Java. In the next chapter we are to introduce Functional Interfaces and lambda expressions. Lambda expressions are a new way to override methods and create anonym classes.

4.3 Functional interfaces

As aforementioned, a functional interface is an interface that specifies only an abstract method. Before proceeding, remember that not all **interface methods are abstract**. Starting with JDK 8, an interface might have one or more default methods. **Default methods are not abstract**. **Neither are static or private interface methods**. Therefore, an interface method is abstract only if it does not specify an implementation.

This means that a functional interface can include default, static, or private methods, but in all cases, it must have a single, single abstract method. Because non-default, non-static, and non-private interface methods are implicitly abstract, there is no need to use the abstract modifier (although you can specify it, if desired).

In short, they are the tools to support lambda expressions in order to carry out this new paradigm in Java. In the case of interfaces, the concept is to represent a unique type of operation as we will see later.

Look at interfaces and classes that implement those interfaces that they have already existed in previous versions of Java or with a single method such as **Listener**, **Runnable**, **Callable**, or **Comparable**. Can we create and anynym class from these interfaces? Of course.

```
Runnable thread = new Thread("New Thread") {  
    public void run(){  
        System.out.println("run by: " + getName());  
    }  
};
```

```
new ActionListener() {  
    public void actionPerformed(ActionEvent arg0) {  
        System.out.println("Botón pulsado:");  
    }  
});
```

We have not addressed yet lambda expressions. One of their functions is override functional interface methods. Can we declare or overwrite **these interfaces** and objects anonymously with a Lambda **expression**? Yes, it is.


```
Runnable threadLambda = () -> System.out.println("Runnable ");
```

This code

```
Runnable thread = new Thread("New Thread") {  
    @Override  
    public void run() {  
        System.out.println("run by: " + getName());  
    }  
};
```

And this code perform the same action. The run method has been overridden resulting in an anonym class Runnable typed.

```
Runnable threadLambda = () -> System.out.println("run by: " + getName());
```

Those prior interfaces are not functional interfaces perse. Nevertheless, they can behave like Functional interfaces in Java. Their name, legacy interfaces.

As we have already described it, A functional interface is an interface that only offers an abstract method. We can tag them with the functional interface tag `@FunctionalInterface`. Although it is optional, it is recommendable.

Here an example of a Functional Interface. Even if the interface includes three methods, only one of them is abstract: `double operation(double a, double b);`

The other two are default:

```
default double identity(double num) {  
    return num;  
}
```

And static:

```
public static int transformToInteger(double num) {  
    return (int) num;  
}
```

You can appreciate that they have body. We can implement methods in interfaces of these two types.

```
@FunctionalInterface
interface MyFirstFunctionalInterface {

    double operation(double a, double b);
    default double identity(double num) {

        return num;
    }

    public static int transformToInteger(double num) {

        return (int) num;
    }
}
```

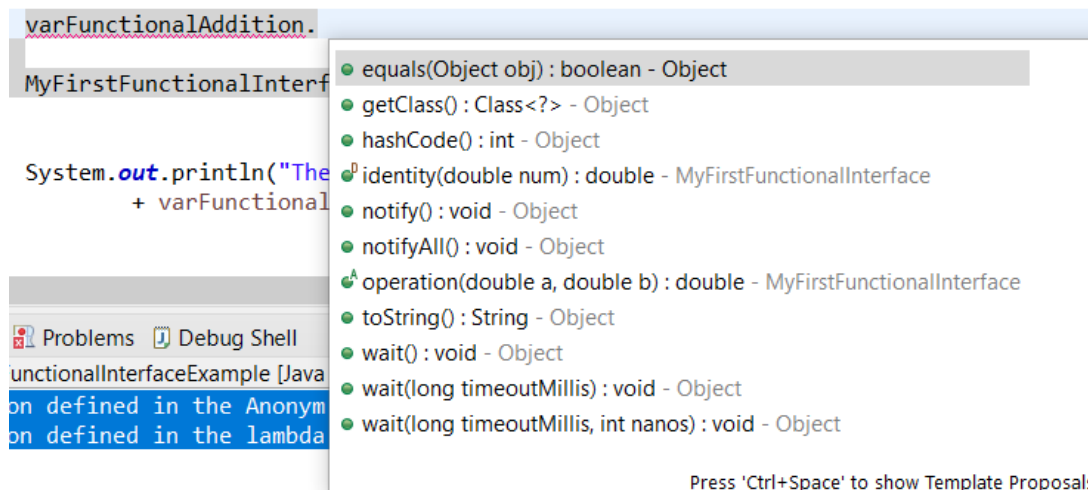
To conclude this exposition, any class that implements this Interface only has to override one method, operation.

Having explained functional interfaces, look at how we create the anonym class, since we follow the same pattern as previously

```
MyFirstFunctionalInterface varFunctionalAddition = new MyFirstFunctionalInterface() {

    @Override
    public double operation (double a, double b) {
        // TODO Auto-generated method stub
        return a+b;
    }
};
```

Now we have and Anonym object stored int the varFunctionalAddition. The object follows the structure defined by the Interface. It inherits methods from the Object class and also the default method defined in MyFirstFunctionalInterface:



You can call the overridden operation method, which will perform an addition:

```
System.out.println("The operation defined in the Anonym class gives as a result:"
    + varFunctionalAddition.operation(5, 7));
```

What is new is the use of the lambda expression. As abovementioned, lambda expressions override the abstract method of an interface. Therefore, if we call the varFunctionalProduct method operation, you can infer that it will calculate the product of these two numbers passed as arguments to the method operation.

```
MyFirstFunctionalInterface varFunctionalProduct = (x,y)-> x*y;
```

```
System.out.println("The operation defined in the lambda expression gives as a result:"
    + varFunctionalProduct.operation(5, 7));
```

Do not panic, we will explain it in the subsequent section.

The execution console outcome for this program:

```
The operation defined in the Anonym class gives as a result:12.0
The operation defined in the lambda expression gives as a result:35.0
```

FunctionalInterfaceExample.java

```
package introductionfunctionalprogramming;
```

```
public class FunctionalInterfaceExample {

    @FunctionalInterface
    interface MyFirstFunctionalInterface {

        double operation (double a, double b);
        default double identity(double num) {

            return num;
        }

        public static int transformToInteger(double num) {

            return (int) num;
        }
    }

    public static void main(String[] args) {

        MyFirstFunctionalInterface varFunctionalAddition = new
        MyFirstFunctionalInterface() {

            @Override
            public double operation (double a, double b) {
                // TODO Auto-generated method stub
                return a+b;
            }
        };

        System.out.println("The operation defined in the Anonym class gives as a result:"
            + varFunctionalAddition.operacion(5, 7));

        MyFirstFunctionalInterface varFunctionalProduct = (x,y)-> x*y;

        System.out.println("The operation defined in the lambda expression gives as a result:"
            + varFunctionalProduct.operation(5, 7));

    }
}
```

4.4 *Lambda expressions.*

The lambda expression is based on a syntax element and an operator that differ from what you saw in the previous topics. The operator, sometimes called a lambda operator or arrow operator is `->`.

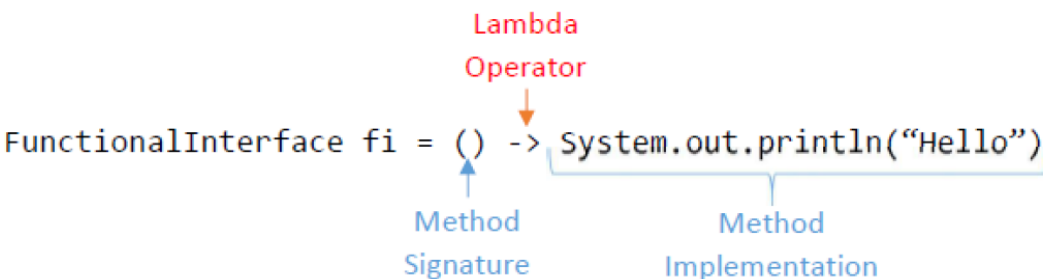
This operator divides a lambda expression in two parts:

- **The left side** specifies the arguments required by the lambda expression.
- **On the right side** is the lambda body, which specifies the actions of the lambda expression.

Java defines two types of lambda bodies. The first type consists of a single expression, and the other type consists of a statement block or block of code. We'll start with lambdas that define the single expression.

Lambda single expression

In the following figure you can see the different parts of a lambda expression:



Let us go back to the former example:

```
@FunctionalInterface
interface MyFirstFunctionalInterface {

    double operation(double a, double b);
    default double identity(double num) {

        return num;
    }
}
```

```
MyFirstFunctionalInterface varFunctionalProduct = (x,y)-> x*y;
```

Going over the lambda expression that implemented the method operation, we can conclude that the lambda expression signature matches the Functional interface signature. The lambda expression arity differs from the operation method arity in the type declaration. Whereas the lambda expressions do not define types, the interfaces does define them. Moreover, the method has a name, yet the lambda expression do not. Let us solve this riddle.

Lambda expression as are based on lambda calculus. This branch of calculus is characterized by the definition of anonymous functions. Traditionally in Math, functions are defined as $f(x) = x+1$;, $\text{name}(\text{variable}) = \text{math expression}$. On the contrary, lambda calculus allows functions that are nameless, **anonymous functions**. To declare them, lambda calculus provides two notations:

- a. $\lambda x. x+1$
- b. $x \rightarrow x+1$

Some of programming language, including Java, use the second notation to declare their own anonymous functions.

In the preceding example the functional interface `MyFirstFunctionalInterface` act as a template for classes in general. Moreover, it is a template for all the lambda expressions you can declare following the interface pattern. In doing so, you do not need to declare types in the lambda expressions considering that types are already declared in the interface. The compiler infers that (x,y) arguments are double typed since the abstract method operation, type them as double: `double operation(double a, double b);`.

Following this line of thought, x pairs with double a, y pairs with double b. There is something left in the equations, and that is the return type. The return type for the method operation is double. Although there is not return statement in the lambda expressions, the compiler assumes that the lambda expression returns the java statement `x*y` result. The **return sentence is implicit in the code**. This pattern matching works for single line lambda expressions. As we will demonstrates later, for block statement lambdas, we need to add a return statement. Let us show it with the following lambda you may add at the end of the `FunctionalInterfaceExample`:

Lambda block expressions

The right subsequent block lambda expression is a block statement. In this case, it is mandatory to explicitly declare the return sentence at the end of the block statement as long as the method that overrides defines a return type.

```
MyFirstFunctionalInterface varFunctionalblock = (x,y)-> { x=2*x ; return x*y; };
```

The syntax for a lambda block expression is:

```
(param1, param2..., paramn) -> {  
    Statement 1;  
    Statement 2  
    ...  
    Statement n;  
    Return statement ; only if the functional interface defines a return type.  
}
```

The evidence presented teach us that a lambda expression is equivalent to a function. Differences emerge from the anonymous nature of lambdas, and the implicit declaration of types, which is inferred from the Functional interface related to the lambda.

Another example will be displayed to complete this section. We will try to write the code for a lambda expression that displays in the console only the subset of the even numbers from the n first numbers.

The first step to take is declaring the functional interface needed to create a variable suitable for the lambda. As we pointed out, the lambda does not need to return a value, since its work is displaying even numbers, so the abstract method in the interface returns void, **void** displayEvenNumbers(**int** n);.

Resulting from the fact that we need to print in the console the subset of the even numbers, from the n first numbers, the method requires n as a parameter.

```
@FunctionalInterface  
interface EvenNumbers {  
    void displayEvenNumbers(int n);  
}
```

The next step is writing the algorithm in the lambda expression. We declare a `EvenNumbers` interface variable. After that, we assign the lambda. The lambda receives `n` as an argument. The for loop iterates `n` times and only print the number if `i%2==0`, which stands for the rest of dividing the `i` variable by 2 is zero. Remember that an even number is a number divisible by 2.

```
EvenNumbers evenLambda = (n) -> {  
    for (int i=0; i<=n ; i++) {  
        if (i%2==0)  
            System.out.println("The number " + i + " is even.");  
    }  
};
```

You can realize that this block lambda does not have a return statement since the `displayEvenNumber` method returns void. And then, to test this lambda, we need to make a call to the `displayEvenNumber` method.

```
evenLambda.displayEvenNumbers(10);
```

Here a runtime result:

```
The number 0 is even.  
The number 2 is even.  
The number 4 is even.  
The number 6 is even.  
The number 8 is even.  
The number 10 is even.
```

Practice

The idea of this practice is making the Functional Interface generic and try the algorithm for Long and Float.

```
@FunctionalInterface  
interface EvenNumbers<T> {  
    void displayEvenNumbers(T n);  
}
```


EvenNumbersExamples.java

```
package generic;
@FunctionalInterface
interface EvenNumbers {

    void displayEvenNumbers(int n);

}

public class EvenNumberExamples {

    public static void main(String[] args) {

        EvenNumbers evenLambda = (n) -> {

            for (int i=0; i<=n ; i++) {

                if (i%2==0)
                    System.out.println("The number " + i + " is even.");

            }

        };

        evenLambda.displayEvenNumbers(10);

    }

}
```

4.5 Predefine Functional Interfaces in Java

In our previous programs, to define a lambda we needed to declare and define a Functional Interface. To avoid that, the Java API offers predefined functional interfaces. It saves the developer time and reduce the amount of code for our programs. In this section we will present some of the most useful Java functional interfaces.

We will review some of the most useful predefined Functional Interfaces in Java. The package `java.util.function` contains all this functional interfaces, this we need to import them whether we utilize them in our programs.

The official documentation for this package is in here:

<https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/util/function/package-summary.html>

4.5.1 Function composition

One of the key points of functional or declarative programming is function composition. This concept derives from the mathematical concept of composition. Most of the Predefined Functional Interfaces offer the possibility of composition. Considering that, we should explain the concept from a mathematical point of view.

Let us define the next functions

$F(x) = x + 1$

$G(x) = 2x$

$F(2)$ could be evaluated as 3 ($2 + 1$)

$G(2)$ values is 4 (2×2)

The mathematical composition lets mathematicians combine functions. Then if we compose

$F \circ G(X)$ the mathematical evaluation of $F \circ G(2)$ should work as:

First We evaluate the most inner function $G(x)$, $G(2)$, the return value is 4. We use that 4 as an input for $F(x)$ in the composition, then we must evaluate $F(4) = 5$. As a result, the composition of $F \circ G(2)$ returns as a value a 5.

What if we compose the functions in the opposite order $G \circ F(x)$?

What would be the return value of $G \circ F(3)$?

4.5.2 Predicate Functional Interface

This interface is used to apply selection or filtering operations. Predicates in Java are

Ladera

implemented with interfaces. `Predicate<T>` is a generic functional interface that represents a single-argument function that returns a Boolean value. It is located in the `java.util.function` package. Contains a `test(T t)` method that evaluates the predicate given the argument. This is the method to override by the lambda expression.

Represents the type of operation we would define as conditional on the parameter(s) it receives. It is the programmer duty to define that operation using lambda expressions or anonymous functions.

The interface definition in the Java API is `Predicate<T>`. It can be inferred here that the predicate is generic interface. It works with different types.

`@FunctionalInterface`

`public interface Predicate<T>`

It offers the subsequent methods:

| | |
|--|--|
| default <code>Predicate<T></code> | <code>and(Predicate<? super T> other)</code> Returns a composed predicate that represents a short-circuiting logical AND of this predicate and another. |
| static <T> <code>Predicate<T></code> | <code>isEqual(Object targetRef)</code> Returns a predicate that tests if two arguments are equal according to <code>Objects.equals(Object, Object)</code> . |
| default <code>Predicate<T></code> | <code>negate()</code> Returns a predicate that represents the logical negation of this predicate. |
| default <code>Predicate<T></code> | <code>or(Predicate<? super T> other)</code> Returns a composed predicate that represents a short-circuiting logical OR of this predicate and another. |
| boolean | <code>test(T t)</code> |

In the table, it is pinpointed that the only abstract methods to implement is `test`, due to the Functional Interface nature of predicate. The other methods are default and offer extra functionality to the `Predicate`, as for example, the ability to combine some predicate variables. The combination of functional interfaces is the result of applying the mathematical concept of function composition, we will detail later.

In the next chunk of code, we are defining a `Predicate` that verifies if a number is greater than 10. We also declare another one that checks if a number is less than 20. Finally, we combine both with methods offered by the predicate interface to combine `Predicate` interfaces.

As abovementioned, the `Predicate` interface belongs to the `java.util.function` package. To

Ladera

use it in our program we need to import it.

```
import java.util.function.Predicate;
```

Secondly, we are defining a variable `Predicate<Integer> greaterThanTen`. The aim of the implementation is creating a lambda that test if a number is greater than 10. Thence, we need to declare a numeric predicate, such as `Integer`. In the inline declaration right side we have the lambda expression `(n)-> n>10;`. It receives an `Integer` number as a parameter and boolean type is returned since we are using a comparison operator.

Finally, we are testing the predicate using the test method. The test method receives an `Integer` as we have declared the predicate as `Integer Predicate<Integer>`. Moreover, when we call the test method the lambda `(n)-> n>10;` because the lambda overrides the test method in `Predicates` functional interfaces.

```
int number = sc.nextInt();

Predicate<Integer> greaterThanTen = (n)-> n>10;

System.out.println("Is it the number " + number + " greater than 10? " +
    greaterThanTen.test(number));
```

The execution result is:

Please type an integer number

13

Is it the number 13 greater than 10? True

For 13 the, test method returns true, following the lambda implementation

`(13)->13>10`, and `13>10` is evaluated as true.

Cornell notes:

Comment the next code, and explain how it works

```
Predicate<Integer> lessThanTwenty = (n)-> n<20;

System.out.println("Is it the number " + number + " less than 20? " +
```

```
lessThanTwenty.test(number));
```

PredicateInterface.java

```
package generic;

import java.util.Scanner;
import java.util.function.Predicate;

public class PredicateInterface {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.println("Please type an integer number");

        int number = sc.nextInt();

        Predicate<Integer> greaterThanTen = (n)-> n>10;

        System.out.println("Is it the number " + number + " greater than 10? "+
            greaterThanTen.test(number));

        Predicate<Integer> lessThanTwenty = (n)-> n<20;

        System.out.println("Is it the number " + number + " less than 20? "+
            lessThanTwenty.test(number));

        Predicate<Integer> and = greaterThanTen.and(lessThanTwenty);

        System.out.println("Is it the number " + number + " greater than 10 " +
            " and " + " lower than 20? " + and.test(number));

        Predicate<Integer> or = greaterThanTen.or(lessThanTwenty);

        System.out.println("Is it the number " + number + " greater than 10 " +
            " or " + " lower than 20? " + and.test(number));

    }

}
```

Predicate composition

To complete this subchapter, we are to explain the last part of the code. We are using the default method `and`, to combine to predicates and produce a new one, `Predicate<Integer> and = greaterThanTen.and(lessThanTwenty);`. The `and` method returns a new predicate which results from the `and` logical composition of the two. Hence, providing that we have two predicates, `greaterThanTen` evaluates only if the number is greater than 10, and `lessThanTwenty` evaluates if a number is less than 20, The `and` composition gives as a result a new predicate that evaluates if the number is greater than ten and lower than 20.

```
Predicate<Integer> and = greaterThanTen.and(lessThanTwenty);

System.out.println("Is it the number " + number + " greater than 10 " +
    " and " + " lower than 20? " + and.test(number));
```

Here an execution:

```
Is it the number 14 greater than 10 and lower than 20? true
```

Cornell notes

Comment this code

```
Predicate<Integer> or = greaterThanTen.or(lessThanTwenty);

System.out.println("Is it the number " + number + " greater than 10 " +
    " or " + " lower than 20? " + and.test(number));
```

Practice

Write a predicate that checks if a number is a prime number.

The algorithm for prime numbers is:

```
Read (n)
Integer i=1
boolean primeNumber = true
```

While (i<=n/2)

If (n%i==0)
 primeNumber=false

End While

4.5.3 The Consumer Interface

The Consumer <T> is intended to represent o be the template function with a T-type argument (parameters) that returns void, nothing. This interface will basically consume some input data and perform an action. That is, the interface represents or defines an operation that performs an action and returns no result. The interface offers the abstract method accept to be overridden by lambda expression or anonymous classes.

The declaration of the consumer interface has this form:

Interface Consumer<T>

The consumer interface offers the aforementioned accept method and some default methods for function composition:

```
void                accept (T t)
                    Performs this operation on the given argument.

default Consumer<T> andThen (Consumer<? super T> after)
                    Returns a composed Consumer that performs, in sequence, this operation
                    followed by the after operation.
```

This interface is the blueprint for lambdas that works as procedures. They receive parameters but they do not return any value. Programmers use these kinds of functional interfaces to print in the Console, Graphical Interface, or to write in Files or databases.

Here an example we will try to analyse:

```
,
Consumer<Integer> consumer = i -> System.out.println("Consumer 1 do an operation " +
+i*i);
```

This consumer will run what is in the lambda, the square of the variable i.

```
System.out.println("Now we execute the first consumer");
consumer.accept(7);
```

The execution result:

Now we execute the first consumer
Consumer 1 do an operation 49

```
Consumer<Integer> consumerWithAndThen =  
    consumer  
    .andThen(i -> System.out.println("Consumer With and Then: Explain the  
operation we print he square of :" + i ));
```

We build the second consumer from the first one consumer and the method `andThen`. This new consumer now has two operations. When we call to the `accept` method of `consumerWithAndThen`, first it will execute the `consumer lambda`. After that, it will execute the second lambda, the new we have added. As it is illustrated in the execution below, the new brand `consumerWithAndThen` is made up to consumers. The original one plus the expression we have added as a parameter for the `andThen` method. Again, we are composing two interfaces, two functions.

Now we execute the second consumer
Consumer 1 do an operation 36 -> first lambda
Consumer With and Then: Explain the operation we print he square of :6 -> second lambda

We must take into consideration that the second lambda it won't be executed until the first one has completed its duty. In programming, we know this behavior as a callback, a function that executes as soon as the function to which it is related has finished its executions.

Finally, to demonstrate that we can create anonymous classes from Functional Interfaces, we have this piece of code. In here, we are overridden the Consumer Method `accept`.

```
Consumer<Integer> consumer3 = new Consumer<Integer> () {  
  
    @Override  
    public void accept(Integer t) {  
        // TODO Auto-generated method stub  
  
        System.out.println("Anonymous consumer");  
    }  
  
};
```


ConsumerInterface.java

```
import java.util.function.Consumer;

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

        Consumer<Integer> consumer = i -> System.out.println("Consumer 1 do an operation "
+ +i*i);
        Consumer<Integer> consumerWithAndThen =
            consumer
                .andThen(i -> System.out.println("Consumer With and Then: Explain the
operation we print he square of :" + i ));

        System.out.println("Now we execute the first consumer");
        consumer.accept(7);
        System.out.println("Now we execute the second consumer");
        consumerWithAndThen.accept(6);

        Consumer<Integer> consumer3 = new Consumer<Integer> () {

            @Override
            public void accept(Integer t) {
                // TODO Auto-generated method stub

                System.out.println("Anonymous consumer");
            }

        };
    }
}
```

Practice

A. We will create a consumer that offers a menu with this options

1. Convert from Farehheit to Celsius
2. Convert from Celsius to Fahrenheit
3. Convert from pounds to kilograms
4. Convert from Kilograms to pounds
5. Convert from Kw to Powerhorse

6. Convert from PowerHorse to Kw

Each option will read a value from the console and will convert it to the target measure. Results will be display on the console.

- B. Implementing a nested loop, could you create a consumer that show the list of the n first prime numbers.

4.5.4 Supplier Interface

The supplier interface performs the opposite function to Consumers. It allows to override with a lambda that receive no parameter and returns a value. The supplier definition in Java is:

```
@FunctionalInterface  
public interface Supplier<T>
```

The method to be overridden is get at state it in the table below:

| Modifier and Type | Method and Description |
|-------------------|---------------------------------|
| <u>T</u> | <u>get</u> () Gets a result. |

Heed your attention to the get method. It returns a generic type T and it does not declare any argument. Java does not offer composition of Functional Interfaces for Suppliers. The next example addresses the supplier description. It is a remarkably simple Functional Interface. Suppliers are aimed to offer values to programmers, following different schemes. These values can be numbers in a Series, might be data from a File or a Database.

We have created two suppliers object in the example.

The first one read a name from the console, and it returns this name:

```
Supplier<String> supNombre = ()->{  
    System.out.println("Could you introduce your name, please?");  
    return sc.nextLine();  
}
```

Notice how the left side of the lambda has no parameter in its definition ()->

The second supplier returns a random number from 0 to twenty. We use the Random class from java.util so as to generate the random number. The method we call is nextInt(20) which generates a random int from the range of 0 and 20.

If we look over the example, we have created the object but we have not assigned it to a variable. Instead, we are calling to the object method straightforward. It is a common practice in Java functional.

```
new Random().nextInt(20);
```

This program illustrates a quite important feature of lambdas. Let us go over the Scanner sc variable. Providing that you would like to make use of the variable inside the lambda, sc must be declared final. This results from the lambda specification, and lambdas being an object, given that lambdas are not allowed to modify variables that are not in their scope. As you can see sc is out of the lambda block, out of its scope.

The compiler will not let use the sc variable inside the lambda unless it is defined using the final modifier, which makes it immutable, which means that it cannot be modified. The purpose of this characteristic is to avoid side-effects and introduce immutability to our lambdas. These two features are desired characteristics of functional programming, we will introduce in the next chapter. On the contrary, you can modify in the lambda container object properties following the Object-Oriented specification of Java. Yellow highlighted, the property than can be modified. In blue the local variable that must be declared final to be utilized in the lambda block.

```
private static int i=0;
```

```
final Scanner sc = new Scanner(System.in);
```

```
Supplier<String> supNombre = ()->{
```

```
    i=5;
```

```
    System.out.println("Could you introduce your name, please?");
```

```
    return sc.nextLine();
```

```
};
```

SupplierInterface.java

```
package PredefinedInterfaces;

import java.util.Random;
import java.util.Scanner;
import java.util.function.Supplier;

public class SupplierInterface {

    private static int i=0;
    public static void main(String[] args) {

        final Scanner sc = new Scanner(System.in);

        Supplier<String> supNombre = ()->{

            i=5;
            System.out.println("Could you introduce your name, please?");
            return sc.nextLine();

        };

        System.out.println("Name readed from the console: " +supNombre.get());

        Supplier<Integer> supRandomNumber = () -> new Random().nextInt(20);

        System.out.println("Generate random number: "
+supRandomNumber.get());
    }
}
```

Practice

- C. Based on the example, you may create a supplier that returns an integer random value from 0 to 100. The use the supplier in a loop to show 50 random numbers.

Could you do it with doubles?

Note: the following interfaces are function alike due to the fact that they receive parameters, and they return results. We will study three of those although Java offers more of them: UnaryOperator, Function, BiFunction.

4.5.5 The Function Functional Interface

This predefined function interface purpose is to define a template for lambdas that receive a single parameter and returns a value. Along with the BiFunction and the UnaryOperator show a highly usage since the Java API follow this pattern for so many of their classes, and programmers also code these kinds of lambdas.

The Java declaration for the Function Interface has this structure:

Interface Function<T,R>

The interface provides the following method:

| | | |
|---|---|--|
| default <V> <u>Function</u> < <u>T</u> , V> | <u>andThen</u> (<u>Function</u> <? super <u>R</u> , ? extends V> after) | Returns a composed function that first applies this function to its input, and then applies the after function to the result. |
| <u>R</u> | <u>apply</u> (<u>T</u> t) | Applies this function to the given argument. |
| default <V> <u>Function</u> <V, <u>R</u> > | <u>compose</u> (<u>Function</u> <? super V, ? extends <u>T</u> > before) | Returns a composed function that first applies the before function to its input, and then applies this function to the result. |
| static <T> <u>Function</u> <T, T> | <u>identity</u> () | |

The method we must override with the lambda expression is R apply (T t). If we look closer the Interface declaration, it defines two generic types <T,R>. T is the entry

Ladera

parameter Type. R is the returned value. In this case, to declare a variable Function Typed you need to introduce the types enclosed in the diamond operator, i.e. `Function<Integer, Double>`; In this scenario the `apply` method would receive an Integer parameter and it would return a Double value.

Let us set an example.

```
Function<Integer,Double> squareRoot = (n)-> Math.sqrt(n);
```

This `squareRoot` variable stores a lambda that overrides the `Function` `apply` method. Following the `Function<Integer,Double>` interface declaration, we can say that the lambda receives an Integer and returns a Double. Furthermore, the lambda code returns the square root of the passed number.

To execute this lambda code, we call the `apply` method passing an integer, 25.

```
System.out.println( " Square root results in " + squareRoot.apply(25));
```

The result is a Double, 5.0

Square root results in 5.0

This is the simplest version since we have made use of the `apply` method. There are another two interesting default methods for the `Function` Interface, `andThen`, and `compose`. They are intended to carry out function composition for `Function` interfaces.

The `function1` variable receives a String and returns the uppercase version of this string.

```
Function <String,String> function1 = (s-> s.toUpperCase()) + " Function 1 to uppercase.");
```

```
System.out.println( " function1 value " + function1.apply(stringExample));
```

Here the execution:

function1 value MI STRING AS A PARAMETER Function 1 to uppercase.

The `function1` variable receives a String and returns the same string but the outer white spaces have been erased.

```
Function <String,String> function2 = (s->s.trim()) + " Function 2 trim.");
System.out.println( " function2 value " + function2.apply(stringExample));
```

Here the execution:

function2 value Mi String as a parameter Function 2 trim.

FunctionInterface

The interface function3 combine or compose the two former interfaces. We have already introduced the andThen method for other interfaces. It is known that the andThen works as a callback. Once the function1 returns a result, this result is passed as a parameter to the function2 lambda. Until the first function does not end, the second is not executed. Moreover, the result, the outcome of function1 is the input of function2. This is the result `function3.apply(stringExample)` **will return, in this order.**

```
Function <String,String> function3 = function1.andThen(function2);
System.out.println( " function3 value " + function3.apply(stringExample));
```

In the example output you can see how Function 1 has been evaluated first.

```
function3 value MI STRING AS A PARAMETER  Function 1 to uppercase. Function 2 trim.
```

And finally, the compose method. We have already described function composition. In the composition of two math functions, FoG(x), the first to be evaluated is G(x), the most inner function. The output of G(x) will be the input of F(x) the outer function. The compose method works following this pattern. As a result, in the next example function2 will be executed first. The outcome of function2, will be the input of function1. This is the result `function4.apply(stringExample)` **will return in this order.**

```
Function <String,String> function4= function1.compose(function2);
System.out.println( " function4 value " + function4.apply(stringExample));
```

In the example output you can see how Function 2 has been evaluated first.

```
function4 value MI STRING AS A PARAMETER  FUNCTION 2 TRIM. Function 1 to uppercase.
```

FunctionInterface.java

```
package PredefinedInterfaces;

import java.util.Scanner;
import java.util.function.Function;

public class FunctionInterface {
```

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

    String stringExample=" Mi String as a parameter ";

    Scanner sc = new Scanner(System.in);

    Function<Integer,Double> squareRoot = (n)-> Math.sqrt(n);

    System.out.println( " Square root results in " + squareRoot.apply(25));

    Function <String,String> function1 = (s-> s.toUpperCase() + " Function 1
to uppercase.");

    System.out.println( " function1 value " + function1.apply(stringExample));

    Function <String,String> function2 = (s->s.trim() + " Function 2 trim.");
    System.out.println( " function2 value " + function2.apply(stringExample));

    Function <String,String> function3 = function1.andThen(function2);
    System.out.println( " function3 value " + function3.apply(stringExample));

    Function <String,String> function4= function1.compose(function2);
    System.out.println( " function4 value " + function4.apply(stringExample));

}

}
```

4.5.6 The Unary Operator Interface

The Unary Operator is almost identical to the Function. The only remarkable different is that it returns the same Type as it receives as parameter given that it extends from the Function interface. Consequently, you do not need to define the Type twice.

The form of the unary operator is:

```
public interface UnaryOperator<T>
extends Function<T,T>
```

And the methods offered by this Interface are the same as the Function interface provides the methods: apply, andThen, compose and identity. It could be claimed that it is a specialization of the former.

Here an example, where long is the Type received as a parameter, and the returned type as well.

```
UnaryOperator<Long> factorial
```

UnaryOperatorExample.java

```
package PredefinedInterfaces;

import java.util.function.UnaryOperator;

public class UnaryOperatorExample {

    public static void main(String[] args) {

        UnaryOperator<Long> factorial = (n) -> {

            Long res= 1L;
            for (int i = 1; i<=n ;i++) {

                res= res*i;
            }

            return res;
        };

        System.out.println("The factorial of 5 is: " + factorial.apply(5L));
    }
}
```

Activity

Add to former example a Long Unary Operator powerOfTwo that calculates the power of the number. After that, combine it with the factorial Unary to get a new UnaryOperator:

- a. Using the andThen method

Ladera

- b. With the compose method.

Show the different results in the console.

4.5.7 Bifunction functional interface

This interface is a function alike interface that receives two generic types as an input, and the third one would be the returned value.

Here is the form of the Bifunction:

```
@FunctionalInterface
public interface BiFunction<T,U,R>
```

The method to be overridden by the lambda expression is `R apply(T t, U u)`. In total, this interface offers these two methods: `andThen` and `apply`.

| | | |
|---|---|---|
| default <V> <u>BiFunction</u> < <u>T</u> , <u>U</u> ,V> | <u>andThen</u> (<u>Function</u> <? super <u>R</u> ,? extends V> after) | Returns a composed function that first applies this function to its input, and then applies the after function to the result. |
| <u>R</u> | <u>apply</u> (<u>T</u> t, <u>U</u> u) | Applies this function to the given arguments. |

Here another example:

The next interface receives two integers as a parameter and return a Double.

```
BiFunction<Integer, Integer, Double> floatDiv
```

```
package PredefinedInterfaces;
```

```
import java.util.function.BiFunction;
```

```
public class BifunctionInterfaceExample {
```

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

```
        BiFunction<Integer, Integer, Double> floatDiv = (a,b) -> (double) a/b;
```

```

        BiFunction <String, String, String > concatUpperCase = (s1,s2)-> (s1
+s2).toUpperCase();

        System.out.println ("Decimal division of 5 and 7:" + floatDiv.apply(5, 7));

        System.out.println ("concatUpperCase interface:" +
concatUpperCase.apply("String 1", "String 2"));

    }

}

```

4.5.8 Primitive types in functional interfaces

Let's look at different types of specialization of predefined functional interfaces seen in the previous chapter. For almost all primitive types in Java we have their Functional Interface specialization. If you pay attention to the former examples of functional interfaces, all of them receive Generic types. **Generic Types cannot be primitive types**. So as to solve this problem, Java offers Functional interfaces that return or receive primitive types, avoiding the need of pass them as generic types, which is not permitted.

Because a **primitive type cannot be a generic type of argument**, there are versions of the Function interface for the most commonly used primitive types **double, int, long**, and **their combinations** in argument types and return type:

1. *IntFunction, LongFunction, DoubleFunction*- **Arguments** are of specified type, the return type is parameterized.
2. *ToIntFunction, ToLongFunction, ToDoubleFunction*: Return types are of **specified type**,arguments are parameterized.
3. *DoubleToIntFunction, DoubleToLongFunction, IntToDoubleFunction, IntToLongFunction, LongToIntFunction, LongToDoubleFunction* - Have an argument and **return type** defined as **primitivetypes**, as specified by their names.

In the next example we use some of these specializations.

The interface `IntFunction`, receives a primitive type `int` and returns a Generic type, `String` in this case.

```
IntFunction<String> convertToIntString = (i)-> String.valueOf(i);
```

The interface `ToIntFunction`, receives a Generic Type, `String`, and returns a `int` primitive type.

```
ToIntFunction<String> convertToStringInt = (s -> Integer.valueOf(s));
```

Notice how always we define the Generic type in the operator diamond. The primitive type is inferred from the Functional interface name.

In the last example there is no generic type passed as a parameter. Both `int` and `double` are primitive types.

```
IntToDoubleFunction convertToIntADouble = (i -> Double.valueOf(i));
```

All in all, Java includes these specializations to allow programmer to use primitive types in their lambda expressions.

`PrimitiveTypeSpecialization.java`

```
package PredefinedInterfaces;

import java.util.Scanner;
import java.util.function.IntFunction;
import java.util.function.IntToDoubleFunction;
import java.util.function.ToIntFunction;

public class PrimitiveTypeSpecialization {

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

```
        Scanner miScanner = new Scanner(System.in);
```

```
        System.out.println("Introduce an int number");
        int numero1 = miScanner.nextInt();

        IntFunction<String> convertToIntString = (i)-> String.valueOf(i);
        ToIntFunction<String> convertToStringInt = (s -> Integer.valueOf(s));
        IntToDoubleFunction convertToIntADouble = (i -> Double.valueOf(i));

        String cadena = convertToIntString.apply(numero1);

        System.out.println("Convert from type primitive int " + numero1 + " to
String " + cadena);

        int numero2 = convertToStringInt.applyAsInt(cadena);

        System.out.println("Convert from string " + cadena + " to primitive
type " + numero2);

        Double numerodecimal = convertToIntADouble.applyAsDouble(numero1);

        System.out.println("Convert from primitive int" + numero1 + " to primitive
type double " + numerodecimal);
    }
}
```

5 Functional programming

Categorization of Programming Paradigms

Of course, functional programming is not the only programming style in practice. Broadly speaking, programming styles can be categorized into imperative and declarative programming paradigms:

The **imperative approach** defines a program as a sequence of statements that change the program's state until it reaches the final state. Procedural programming is a type of imperative programming where we construct programs using procedures or subroutines. One of the popular programming paradigms known as object-oriented programming (OOP) extends procedural programming concepts.

In contrast, the **declarative approach** expresses the logic of a computation without

Ladera

describing its control flow in terms of a sequence of statements. Simply put, the declarative approach's focus is to define what the program must achieve rather than how it should achieve it. Functional programming is a sub-set of the declarative programming languages.

Functional Programming

Functional programming is a programming paradigm in which we try to bind everything in pure mathematical functions style. It is a declarative type of programming style. Its central focus is on “what to solve” in contrast to an imperative style where the main focus is “how to solve”. It uses expressions instead of statements. An expression is evaluated to produce a value whereas a statement is executed to assign variables. Those functions have some special features discussed below.

Functional Programming is based on Lambda Calculus:

Lambda calculus is framework developed by Alonzo Church to study computations with functions. It can be called as the smallest programming language of the world. It gives the definition of what is computable. Anything that can be computed by lambda calculus is computable. It is equivalent to Turing machine in its ability to compute. It provides a theoretical framework for describing functions and their evaluation. It forms the basis of almost all current functional programming languages.

Fact: Alan Turing was a student of Alonzo Church who created Turing machine which laid the foundation of imperative programming style.

Programming Languages that support functional programming: Haskell, JavaScript, Scala, Erlang, Lisp, ML, Clojure, OCaml, Common Lisp, Racket.

Functional programming languages are categorized into two groups, i.e. –

- Pure Functional Languages – These types of functional languages support only the functional paradigms. For example – Haskell.
- Impure Functional Languages – These types of functional languages support the functional paradigms and imperative style programming. For example – LISP.

Functional Programming Advantages

- Bugs-Free Code – Functional programming does not support state, so there are no side-effect results, and we can write error-free codes.

- Efficient Parallel Programming – Functional programming languages have NO Mutable state, so there are no state-change issues. One can program "Functions" to work parallel as "instructions". Such codes support easy reusability and testability.
- Efficiency – Functional programs consist of independent units that can run concurrently. As a result, such programs are more efficient.
- Supports Nested Functions – Functional programming supports Nested Functions.
- Lazy Evaluation – Functional programming supports Lazy Functional Constructs like Lazy Lists, Lazy Maps, etc.

Building on from the idea of declarative programming, functional programming seeks to solve algorithm problems using function calls. The next example will try to illustrate differences about imperative and declarative programming:

Once we have read the number to which we want to calculate the third power, the imperative solution relies on variable declaration and assignation to solve the problem.

```
//Imperative solution
double result = number*number*number;
System.out.println("Third Power of "+ number + " is " + result);
```

On the other hand, functional programming tries to avoid variable declaration and assignation, which is the main cause of bugs and errors in our programs. As it is showed in the example, in one line we solve the problem with function or method calls

```
//Declarative and functional solution

System.out.println("Third Power of "+ number + " is " +
FunctionalProgrammingExample.thirdPower(number));
```

At the same time, methods in functional programming try to skip assigning values to variable as much as possible.

```
public static double thirdPower(double x) {

    return x*x*x;
}
```

FunctionalProgrammingExample.java

```
package introductionfunctionalprogrammin;

import java.util.Scanner;

public class FunctionalProgrammingExample {

    public static double thirdPower(double x) {

        return x*x*x;
    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.println("Write a number to calculate the Third Power");

        double number = sc.nextInt();

        //Imperative solution
        double result = number*number*number;
        System.out.println("Third Power of "+ number + " is " + result);

        //Declarative and functional solution
        System.out.println("Third Power of "+ number + " is " +
            FunctionalProgrammingExample.thirdPower(number));
    }
}
```

5.1 Java 8 and Functional programming

Functional programming in Java has not been easy historically, and there were even several aspects of functional programming that were not even really possible in Java. In Java 8, Oracle tried to facilitate functional programming, and this effort succeeded to some extent. In these **Java functional programming notes** we will go through the basics of functional programming, and which parts of it are feasible in Java.

Functional programming basics

Functional programming includes the following key concepts:

1. Functions are first-class objects
2. Pure functions
3. Higher order functions

Pure functional programming also has a **set of rules** to follow:

1. stateless
2. No side effects.
3. Immutable variables
4. Favor recursion over the loop

These concepts and rules will be explained in the rest of this unit

Even if you do not follow all these rules all the **time**, we can **benefit from functional programming ideas** in our **programs**. Unfortunately, **functional programming is not the right tool for all problems**. Especially the idea of "no side effects" makes it difficult, for example, to **write to a database** (which is a side effect). But the **java Stream API** and **other functional programming qualities** are for daily use by Java programmers.

5.2 Functions are first-class objects/ first-class members of the language

One of the main advantages that Java 8 and functional programming brings to the table is that the functions has become first-class objects or members of the language through the Functional interfaces. Something that is pretty common in some of programming languages such as JavaScript, which is that functions can be store in variables or can be passed as parameters, it was impossible in previous versions of Java.

In this JavaScript excerpt we are passing the myCallback function as a parameter to the function addContact.

```
function addContact(id, callback) {  
    callback();  
}  
  
function myCallback() {  
    alert('Hello World');  
}  
  
addContact (myCallback);
```

In the functional-programming paradigm, functions are **first-class objects** in the language. This means that you can create an "instance" of a function, as can a variable reference to that function instance, as well as a reference to a String, a HashMap, or any other object. Functions can also be passed as parameters to other functions. In Java, methods are not first-class objects. The closest we get are the functional interfaces in Java.

Let's explain it with a simple example. In our `FunctionFirstClassExample` class, we have a functional interface parameter `formula` that applies the received formula, calling the method `apply`. Notice that to execute the function we need to call the `apply` method, `formula.apply(x)`.

```
public Double functionFormula(Double x, Function<Double,Double> formula) {  
    // ...  
    return formula.apply(x);  
}
```

We have also two functions `formulaSquare` and `formulaThirdPower`. These two functions signature `Double x`, and returned type `Double`, match the signature of the Functional Interface `Function<Double,Double> formula`.

```
public Double formulaSquare(Double x) {
```

```
        return x*x;  
    }  
  
    public Double formulaThirdPower(Double x) {  
        return x*x*x;  
    }  
}
```

We can create an object typed `FunctionFirstClassExample`.

```
FunctionFirstClassExample object = new FunctionFirstClassExample();
```

After that, we can pass to the method `functionFormula` a function of this object or other object from a different class, `object::formulaSquare`. To do so, we use **the :: operator**, which allows programmers to reference functions.

In this case, we reference the function through the object, and the syntax for this expression is:

`object_variable_name::methodName`

```
Double result = object.functionFormula(numero, object::formulaSquare);
```

Hence, the `FunctionalInterface` provides the chance to pass functions as parameter, and also store them in a variable.

```
Function<Double,Double> formulaVar = objeto::formulaSquare;
```

Moreover, we can pass as a parameter a lambda expression, what eventually is a anonymous function.

```
result = object.functionFormula(numero, x->x*x*x*x);
```

Finally, we can pass as a parameter a static method, a class method. The syntax for static methods is slightly different since we use the class name to reference a static method:

class_name::methodName

```
FunctionFirstClassExample::formulaSquareRoot
```

```
result = objeto.functionFormula(numero, FunctionFirstClassExample::formulaSquareRoot);
```

FunctionFirstClassExample.java

```
package functionalinterfaceparadigm;

import java.util.Scanner;
import java.util.function.Function;

public class FunctionFirstClassExample {

    public Double functionFormula(Double x, Function<Double,Double> formula) {

        return formula.apply(x);
    }

    public Double formulaSquare(Double x) {

        return x*x;
    }

    public Double formulaThirdPower(Double x) {

        return x*x*x;
    }

    public static Double formulaSquareRoot(Double x) {

        return Math.sqrt(x);
    }

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

```

Scanner sc = new Scanner(System.in);

System.out.println("Write a number in the console");

Double numero = sc.nextDouble();
FunctionFirstClassExample object = new FunctionFirstClassExample();

Double result = objeto.functionFormula(numero, object::formulaSquare);

System.out.println("We calculate the Square of the number: " + result);

Function<Double,Double> formulaVar = objeto::formulaSquare;

result = object.functionFormula(numero, object::formulaThirdPower);

System.out.println("We calculate the third power of the number: " +
result);

result = object.functionFormula(numero, x->x*x*x*x);

System.out.println("We calculate the fourth power of the number: " +
result);

    result = objeto.functionFormula(numero,
FunctionFirstClassExample::formulaSquareRoot);

    System.out.println("We calculate the square root of the number: " +
result);

}

}

```

An execution here:

```

Write a number in the console
25
We calculate the Square of the number: 625.0
We calculate the third power of the number: 15625.0
We calculate the fourth power of the number: 390625.0
We calculate the square root of the number: 5.0

```

5.3 Pure functions. Stateless.

In programming, a pure function is a function that has the following properties:

1. The function always returns the same value for the same inputs.

Ladera

2. Evaluation of the function has no side effects. Side effects refer to changing other attributes of the program not contained within the function, such as changing global variable values or using I/O streams (no mutation of local static variables, non-local variables, mutable reference arguments or input/output streams).

Effectively, a pure function's return value is based only on its inputs and has no other dependencies or effects on the overall program.

Pure functions are conceptually like mathematical functions. For any given input, a pure function must return exactly one possible value.

Like a mathematical function, it is, however, allowed to return that same value for other inputs. Additionally, like a mathematical function, its output is determined solely by its inputs and not any values stored in some other, global state.

Here an example:

```
public class PureFunctionClass{  
  
    public int sum(int a, int b) {  
        return a + b;  
    }  
}
```

Any time you call the sum function will return the same value, i.e sum(5,7) evaluation will be always 12, no matter how many times you call this function.

Notice how the return value of the suma() function depends only on the input parameters. Also note that the sum() has no side effects, which means that it does not modify any state (variables) outside the function or class.

The opposite it can be found in the following example of a non-pure function:

```
public class NotPureFunctionExample{  
    private int value = 1;  
  
    public int sum(int nextValue) {  
        this.value += nextValue;  
        return this.value;  
    }  
}
```

Here, the effect is the opposite. The first time you call this functions `sum(3)` would return 4. Providing that after this calling the property value changes to 4, the second time you call `sum(3)` the returned result from this method calling would be 7. In the end, the function return value depends on the object property value, which changes in each call.

5.3.1 High-order functions

The idea behind High-order functions stands for the employing of functions as arguments. For a function to be a high-order functions, it must comply the following condition: Any parameter in the function signature must be a function, and the return type must be a function likewise. Ergo, the final product is a function that manipulates functions, and generates a function as a result.

In Java, the closest we can get to a higher-order function is a function (method) that takes one or more functional interfaces as parameters and returns a functional interface. Here's an example of a higher-order function in Java:

In this simpler example you can try a higher order function.

```
public Supplier<Double> functionOrdenSuperiorFunction(Supplier<Double> numero,  
Function<Double,Double> function)
```

The `highOrderFunction` takes as a parameter a `Supplier` interface that gives us a number, a `Function` interface, and returns a `supplier` interface. What we are basically doing is allowing lambda expressions or functions to get passed as a parameter and return a lambda function or expression as a parameter

```
public Supplier<Double> highOrderFunction(Supplier<Double> number,  
Function<Double,Double> function) {  
  
    return ( ()->function.apply(number.get()));  
  
}
```

Ladera

The parameters in the `highOrderFunction` signature are Functional interfaces, lambdas, or functions. Likewise, the returned type is a `Supplier<Double>` another function. It is showed in the return statement of this example, how we are returning a function, which is a lambda in this case.

```
package introductionfunctionalprogrammin;

import java.util.Scanner;
import java.util.function.Consumer;
import java.util.function.Function;
import java.util.function.Supplier;

public class HighOrderFunctionExample {

    public Supplier<Double> highOrderFunction(Supplier<Double> number,
Function<Double,Double> function) {

        return ( ()->function.apply(number.get()));

    }

    public static void main(String[] args) {

        Scanner miScanner = new Scanner(System.in);

        System.out.println("Escriba un número decimal");

        Double number1 = miScanner.nextDouble();

        HighOrderFunctionExample highOrder = new HighOrderFunctionExample()
;

        Supplier<Double> supplierNumeric= highOrder.highOrderFunction(()->
number1 ,(x)->x*x);

        System.out.println(" highOrderFunction returns a Supplier function
" + supplierNumeric.get());

    }

}
```


Owing that this is an absolutely important concept in functional programming, we are to introduce another example. In addition, we will employ some of the concepts we have learned, such as function composition and the `::` operator.

The method `addFormulaSelectedAndRound` receives a `Function<Double,Double>` as an argument. It also returns a `Function<Double,Double>`

```
public Function<Double,Double> addFormulaSelectedAndRound(Function<Double,Double>
function) {

    return function.then(HighOrderFunctionExample2::round);

}
```

Its processing consists of combining the formula passed as a parameter, a function, with the `round` function, displayed yellow-highlighted. Finally, we call to this function in the main program.

```
Function<Double,Double> myFormula =
objectHighOrder.addFormulaSelectedAndRound(HighOrderFunctionExample2::square);
```

The `square` function is a static method that suits with the `Function<Double,Double>` definition; Hence, we can pass it as a parameter

```
public static Double square(Double a) {

    return a*a;

}
```

As a result, we have a function that calculate the square and rounds the result in the `Function<Double,Double> myFormula` variable.

We can employ this function wherever in our code, and infinite times. It is a pure function also; it will return the same value in any call. To invoke the function, we use the `apply` method `myFormula.apply(24.2)`.

Ladera

```
System.out.println("The attained function calculates the square of the number" +  
    " and get the number rounded: " + myFormula.apply(24.2));
```

The execution result here:

The attained function calculates the square of the number and get the number rounded:
586.0

HighOrderFunctionExample2.java

```
package introductionfunctionalprogrammin;
```

```
import java.util.function.Function;
```

```
public class HighOrderFunctionExample2 {
```

```
    public Function<Double,Double> addFormulaSelectedAndRound(Function<Double,Double>  
function) {
```

```
        return function.andThen(HighOrderFunctionExample2::round);
```

```
    }
```

```
    public static Double squareRoot(Double a) {
```

```
        return Math.sqrt(a);  
    }
```

```
    public static Double square(Double a) {
```

```
        return a*a;  
    }
```

```
    public static double round(Double a) {
```

```
        return (double) Math.round(a);
```

```
    }
```

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

```

        HighOrderFunctionExample2 objectHighOrder = new
HighOrderFunctionExample2();

        Function<Double,Double> myFormula =

objectHighOrder.addFormulaSelectedAndRound(HighOrderFunctionExample2::square);

        System.out.println("The attained function calculates the square of the
number" +
        " and get the number rounded the number: " + myFormula.apply(24.2));

    }
}

```

5.3.2 Stateless

As mentioned at the beginning of these chapter, a rule of the functional programming paradigm is to have no state. "No state" or "Stateless" is usually understood by any state outside the function. A function may have local variables which contain the temporary state internally, but the function cannot reference any member variables of the class or object to which the function belongs. You must not access external values.

Example of a function that does not use any external state:

```

public class Calculator{
    public int sum(int a, int b) {
        return a + b;
    }
}

```

On the contrary, in the next version of the sum method the problem is that the method depends on the state of the calculator at a certain time, since its calculation lay in the `initVal` value.

```

public class Calculator {
    private int initVal = 5;

    public setInitVal(int val) {

        initVal= val;
    }
}

```

Ladera

```
public int sum(int a) {  
    return initVal + a;  
}  
}
```

This kind of programming practices can lead to bugs and errors in your code. Nevertheless, since Java is an Object-Oriented Programming language, we have to rely on the state of an object to do some calculations. The way to follow should be trying to find a balance or equilibrium between functional and Object-Oriented programming. Therefore, in some context, when necessary, we apply concepts from Object Oriented Design. It should be mandatory in any programming scenario where we need to represent data or behavior, in other words, to do abstractions.

5.3.3 No side effects

Another rule in the functional programming paradigm is that of no side effects. This means, that a function cannot change any state outside of the function. Changing state outside of a function is referred to as a *side effect*.

It is deduced from the previous example. Providing that my state will not change, an undesired result is unlikely to happen in the different function executions. It is rule in the functional programming paradigm. This means that a function cannot change any state outside the function. Changing the outer object state in a function is known as a *side effect*.

The state outside a function refers to both member variables of the function class or object, as well as member variables within parameters to functions, or state in external systems such as file systems or databases.

```
public class Calculator{
    public int sum(int a, int b) {
        return a + b;
    }
}
```

The call sum (5,7) returned value would be always 12. Consequently, it is correct functional programming method. It does not reproduce or cause errors.

```
public class Calculador {
    private int initVal = 5;

    public setInitVal(int val) {
        initVal= val;
    }

    public int sum(int a) {
        return initVal + a;
    }
}
```

In this second scenario, since the sum functions relies on the property value initVal, the object state, it could lead the program to undesirable errors.

5.3.4 Immutable objects

The definition that you can see in Wikipedia states that an immutable object is one whose state cannot be modified after it is created. Or in other words, some variable that cannot change its value.

An immutable object is an object whose state cannot be modified after it is created. In multi-threaded applications that is a great thing. It allows for a thread to act on data represented by immutable objects without worrying what other threads are up to.

It is very common in large and medium size applications that multiple programs reference the same variable. This is the concept introduced above, multithreading or multiprocessing. To a certain extent, we are trying to avoid the side effect of some processes or threads that could modify the same object in a given time.

In Java, it is quite simple to create an immutable object. The way to proceed is to declare a class with private properties, getters and no setters. The property values are fixed or assigned in the constructor. After the object is created, it is impossible to modify its state.

In the following program, the class `ImmutableObjectPerson` possesses private properties. In addition, no set method is declared for these properties. Once the program creates the object, it is clear that it is not possible to modify its properties.

```
ImmutableObjectPerson person = new ImmutableObjectPerson("John", "Doe", 40);
```

Nevertheless, properties can be evaluated from getters, and that results in the program having full access to the object state.

```
System.out.println("Person full name:" + person.getName() + " " + person.getLastName());
```

`ImmutableObjectPerson.java`

```
package functionalinterfaceparadigm;
```

```
public class ImmutableObjectPerson {
```

```
    private String name="";  
    private String lastName="";  
    private int age=0;
```

```
    public ImmutableObjectPerson() {  
    }  
}
```

```
    public ImmutableObjectPerson(String name, String lastName, int age) {  
        super();  
        this.name = name;  
        this.lastName = lastName;  
        this.age = age;  
    }  
}
```

```
    public String getName() {  
        return name;  
    }  
}
```

```
    public String getLastName() {  
        return lastName;  
    }  
}
```

```
    public int getAge() {  
        return age;  
    }  
}
```

```
@Override
public String toString() {
    return "ImmutableObjectPerson [name=" + name + ", lastName=" + lastName +
", age=" + age + "]";
}

public static void main(String[] args) {

    ImmutableObjectPerson person = new ImmutableObjectPerson("John", "Doe", 40);

    System.out.println("Person full name:" + person.getName() + " " +
person.getLastName());

}
}
```

5.3.5 Favor recursion over loops

A fourth rule in the functional programming paradigm is to favor recursion over the loop. Recursion uses function calls to achieve loops, so code becomes more functional.

Another alternative to loops is the Java Streams API. This API is functionally inspired. We'll see you later in the course.

The principle of recursion is the capability of solving computation problems with a function which calls to itself. In the next section, we will look over recursion extensively.

6 Recursion

One of the programming tools that is necessary in functional programming is the concept of recursion. In general, recursion is the process of defining something in terms of itself and is somewhat equivalent to a circular definition. The key component of a recursive method is a declaration that executes a call to itself. Recursion is a powerful control mechanism.

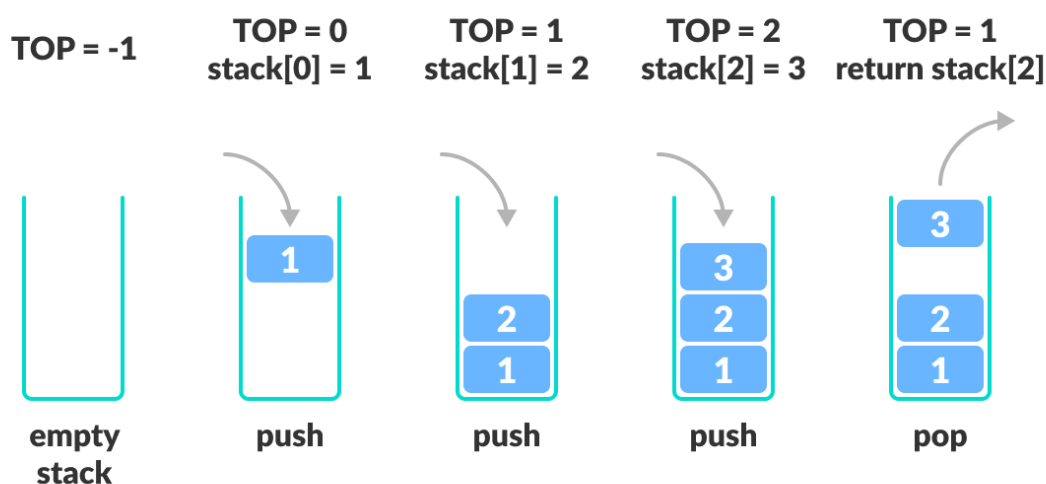
The process in which a function is called directly or indirectly is referred to as

recursion, and the corresponding function is called a recursive function. Using a recursive algorithm, certain problems can be solved quite easily. Examples of such problems are Hanoi Towers (TOH), tree tours, and search algorithms such as quicksort.

6.1 The Java Runtime Stack

6.1.1 ¿What is a stack?

A stack is a java data structure that places elements on top of each other. When we do a `push()` operation, we place an item up. When we make a `pop()`, we take out the element that is above it.



6.2 The Java Runtime Stack

Every time we call a function or method in one of our programs from the main class or program that method will take control of execution in our program. To save its execution and the state of all its local variables and parameters, Java implements, like other compilers, a limited-size runtime stack or Runtime Stack, for each program.

For each process the Java virtual machine, Java Virtual Machine (JVM) will create a runtime stack. Each method will be inserted into the stack. Each entry is labelled as Stack Frame Activation Log.

Ladera

Each time we call a method or function, that Java method or function creates an entry in the stack. This mechanism is used to know in which execution and who has control of the execution of our program at a given time. For each call, a copy of its local variables and parameters is stored in memory.

In the subsequent example, it is obvious how the Java Runtime Stack operates. From the very first moment that the program start executing, the first element to be pushed on the stack is the main() function. The main() method takes control of the program flow, and subsequently, main() makes a call to the first_func() method. As a result, the JVM pushes first_func() to the Runtime stack, and eventually first_func() takes control of the execution.

Besides, first_func() is to invoke second_func(). Again, second_func goes on top of the stack and takes full charge of the program execution. Once second_func ends, the JVM pops the method from the stack. The method first_func() take over again, and as long as it finishes its executions will be withdrawn from the stack. Finally, the flow returns to main program. Once the main program ends, it is also taken out from the stack, and all program resources should be released by the Garbage Collector. Among these resources, which can be files, database connections and so on, the Runtime Stack is included. Once the program resources are freed, the process related to the program would be destroyed.

| (1) | (2) | (3) | (4) | (5) | (6) | (7) |
|----------------------------------|--------|--------------|---------------|--------------|--------|----------------|
| *(Runtime Stack for Main Thread) | | | | | | (*Empty Stack) |
| | | | second_func() | | | |
| | | first_func() | first_func() | first_func() | | |
| | main() | main() | main() | main() | main() | |



```

/**
 *
 */
package com.test.java;

/**
 *

```

Ladera

```
public class RuntimeStackMechanism {  
  
    /**  
     * @param args  
     */  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        first_func();  
    }  
  
    public static void first_func() {  
        second_func();  
    }  
  
    public static void second_func() {  
        System.out.println("Hello World");  
    }  
}
```

The Garbafe Collector

Needless to say that **the Garbage Collector** is one of the many processes that the JVM possesses to offer Java execution capabilities as the Runtime Stack is. Particularly, from the name, it can be inferred that the Garbage Collection deals with finding and deleting the garbage from memory. However, in reality, garbage Collection tracks each and every object available in the JVM heap space and removes unused ones.

In simple words, GC works in two simple steps known as Mark and Sweep:

- **Mark** - it is where the garbage collector identifies which pieces of memory are in use and which are not
- **Sweep** - this step removes objects identified during the “mark” phase

The Garbage collectors provides:

- No manual memory allocation/deallocation handling because unused memory space is automatically handled by GC.
- No overhead of handling Dangling Pointer. A dangling pointer is a variable that reference an object memory address which is not of the correct type. It is not possible in Java.
- Automatic Memory Leak management (GC on its own can't guarantee the full proof solution to memory leaking, however, it takes care of a good portion of it). A memory leak is a type of resource leak that occurs when a computer program incorrectly manages memory allocations

6.3 Recursion

What is the base condition in recursion?

In the **recursive program**, the solution to the base case is **provided** and the solution of the **largest problem** is expressed **in terms of smaller problems**. This ideas concord with the **Divide and Conquer problem solving strategy**.

```
public static Long factorial(long n)
{
    if (n <= 1) // base case
        return 1;
    else
        return n*factorial(n-1);
}
```

In the above example, base case for $n \leq 1$ is defined and larger value of number can be solved by converting to smaller one till base case is reached.

How is a particular problem solved using recursion?

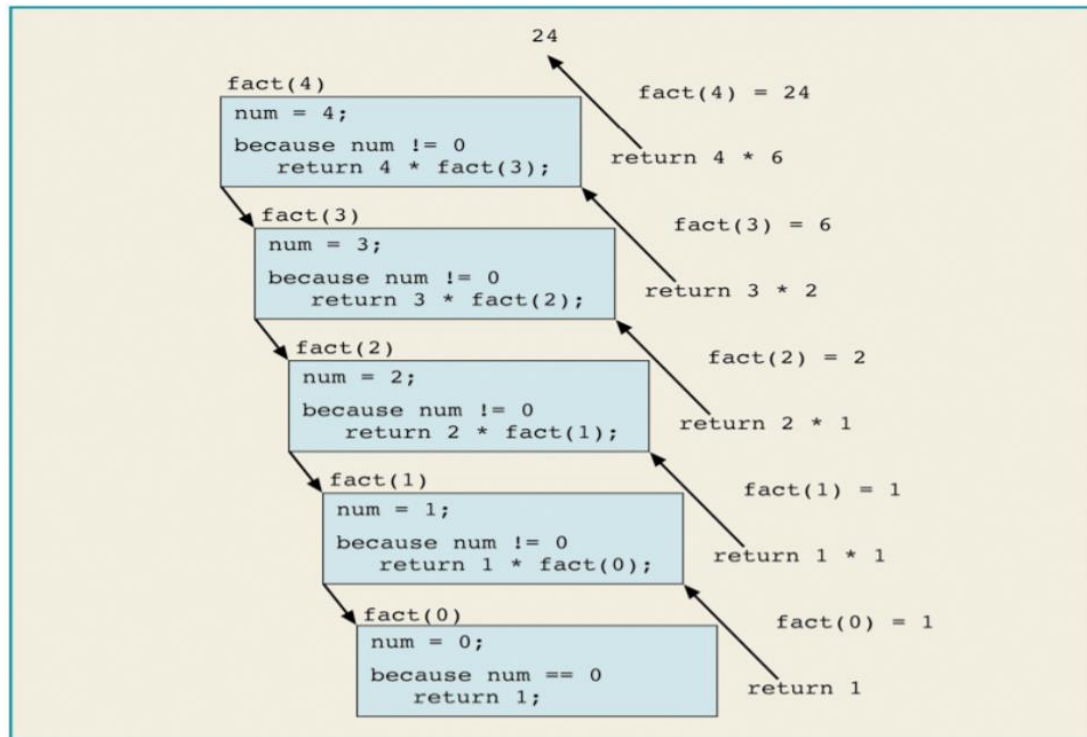
The idea is to represent a problem in terms of one or more smaller problems and add one or more base conditions that stop the recursion. For example, we compute factorial n if we know factorial of $(n-1)$. The base case for factorial would be $n = 0$. We return 1 when $n = 0$.

In the former solution, providing that we call `factorial(4)`, this call would return $4 \times \text{factorial}(3)$. `Factorial(4)` wait until `factorial(3)` is evaluated. Hence, the control flow is in `factorial(3)`. Likewise, `factorial(3)` evaluation returns $3 \times \text{factorial}(2)$. Again, the function evaluation must wait for another function call, `factorial(2)`. The `factorial(2)` invocation, would evaluate $2 \times \text{factorial}(1)$. One more time. `Factorial(2)` is on halt. Now `factorial(1)` is to be evaluated. Nonetheless, the execution has reached the base case. `Factorial(1)` does not depends on any other call since it returns straightforward 1.

The Runtime Stack mechanism should come back the opposite way. Given that `factorial(1)` returns 1 and is popped out of the stack, `factorial(2)` does not need to wait any longer, and can be fully evaluated as $2 \times \text{factorial}(1) = 2 \times 1 = 2$. Consequently, `factorial(2)` is taken out from the stack as it returns 2. It is the turn of `factorial(3)`. At this moment `factorial(3)` call has received the `factorial(2)` value, and thus $3 \times \text{factorial}(2)$, now can be calculated as $3 \times 2 = 6$. And eventually we are at the end of the recursion, since it is `factorial(4)` which recover the flow and executes the return statement $4 \times \text{factorial}(3) = 4 \times 6 = 24$. In the end, `factorial(4)` returns 24 and the program ends. The imperative approach solution for this problem should be a loop. Following the declarative paradigm, programmers use recursion

Ladera

to crack down problems in a different manner.

**FactorialRecursion.java**

```

package recursion;
import java.util.Scanner;

public class FactorialRecursion {

    public static Long factorial(long n)
    {
        if (n <= 1) // base case
            return Long.valueOf(1);
        else
            return n*factorial(n-1);
    }

    public static void main(String[] args) {

```

```

Scanner miScanner = new Scanner(System.in);

System.out.println("Type an integer number");
Long number1 = miScanner.nextLong();

Long result = FactorialRecursion.factorial(number1);

System.out.println("The factorial of " + number1 + " is: "+ result);

}

}

```

Why does the stack overflow error means in recursion?

In the even that **the base case is not reached or is not defined**, the stack overflow problem may **arise**. Let's take an example to understand this. Whether there is no base case or it is not proper defined, we cannot **return or reverse from factorial(1)** and the function would be **infinitely left by calling itself**, until it **overflows the size of the java execution stack**. Provided that the **stack has one million available entries**, in the factorial call number million and one, the stack overflows. The **stackoverflow exception is thrown** in Java.

```

Long factorial(Long n)
{
    // it cause overflow
    //
    if (n == 100)
        return Long.valueOf(1);

    else
        return n*fact(n-1);
}

```

If you call **fact(10)**, you will call **fact(9)**, **fact(8)**, **fact(7)**, and so on, but the parameter will

Ladera

never reach **100, because it is decreasing**. Therefore, the **base case is not reached**. Once these infinite function calls end up the memory assigned to the stack, a **stack overflow error will occur**.

What is the difference between direct and indirect recursion?

A function is called direct recursion providing that it calls the same function. A function is called indirect recursiveRecFun1 whether it calls another new **indirect** FunctionRecFun2. The indirect FunctionRecFun2 will call indirectRecFun1 again until one of **them reaches** the **base case** as in a ping pong game, switching from one function **to the other**. The difference between direct and indirect recursion has been illustrated in Table 1.

Recursividad directa

```
void directaRecFun()
{
    // Some code....

    directaRecFun();

    // Some code...
}
```

Recursividad Indirecta

```
void indirectaRecFun1()
{
    // codigo...

    indirectaRecFun2();

    // codigo...
}

void indirectaRecFun2()
```

```
{  
    // código...  
  
    indirectaRecFun1();  
  
    // código  
}
```

As you can see in indirect recursion, we have two functions. One calls the other while making the problem smaller until you reach the base case in one of them.

What is the difference between queue and header recursion?

A recursive function is queue recursive when the recursive call is the last statement executed by the function.

For example, in the example above the last thing factorial does is calling itself recursively:

return n*factorial(n-1). Tail-recursion

The following example illustrates head recursion. Because the recursive call is not the last function statement.

```
// Instruccion 2  
printFun (test - 1);  
  
System.out.printf("%d ", test);  
return;
```

How is memory allocated to different recursion function calls?

When any function is called from main(), it is allocated memory on the stack. As the recursive function calls itself, the memory of the called function is allocated above the memory allocated to the calling function, and a different copy of the local variables is created for each function call. When the base case is reached, the function returns its value to the function by which it was called. Moreover, this function is deallocated from the stack and the process continues.

Let's take the example of how recursion works by taking a simple function.

```
public class PrintRecursive {
    static void printFun(int test)
    {
        if (test < 1)
            return;

        else {
            System.out.printf("%d ", test);

            printFun(test - 1);

            System.out.printf("%d ", test);
            return;
        }
    }

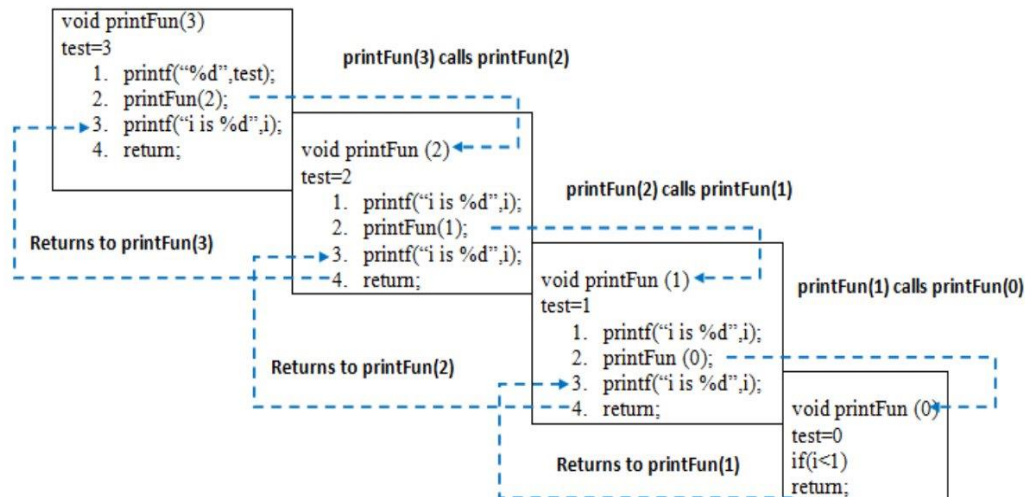
    public static void main(String[] args)
    {
        int test = 3;
        printFuncion(test);
    }
}
```

When `printFun(3)` is invoked, the function is pushed to the stack, and the `test` local variable is initialized to 3. The variable is printed, and thus a 3 is written in the console. Then a call to `printFun(2)` is made. Again, `printFun(2)` is allocated in memory on top of the stack, and a two value is assigned to the local variable `test`. This two values is printed, and call is made to `printFun(1)`. For `printFun(1)`, the local variable `test` value is one. Also one is printed and an invocation to `printFun(0)` is performed. Finally, in `printFun(0)` the program has reached the base case, and the return statements is executed, and `printFun(0)` is deallocated from memory. Up to now the program has written the sequence 3,2,1.

From here on, the program follows the flow in the opposite direction. It returns to

Ladera

printFun(1) which prints 1, finish its executions, and it is deallocated from the stack. The same printFun(2) as it prints two, and printFun(3) which offers a three in the console, reaching the end of the program. After that the comeback sequence it is written 1,2,3 in the console.



What are the disadvantages of recursive programming over iterative programming?

Note that both recursive and iterative programs have the same problem-solving capability, that is, each recursive program can be written iteratively and vice versa is also true. The recursive program has higher space requirements than the iterative program since all functions will remain on the stack until the base case is reached. It also has higher time requirements because function calls are put in and pulled out of the stack. You waste time on this, and you take care of more memory.

What are the advantages of recursive programming over iterative programming?

Recursion provides a clean and easy way to write code. Some problems are inherently recursive such as tree tours, Hanoi Tower, etc. For these problems, it is preferred to write recursive code. We can also write these codes iteratively with the help of a stack data structure.

6.4 Practice

1. Calculate the Fibonacci series recursively.

Fibonacci's succession has been known for thousands of years, but it was Fibonacci (Leonardo de Pisa) who made it known when using it to solve a problem.

The **first** and **second** terms of the sequence are

$$\alpha_0 = 0$$

$$\alpha_1 = 1$$

The following terms are obtained by adding up the two preceding terms:

The **third** term of the sequence is

$$\alpha_2 = \alpha_0 + \alpha_1 = 0 + 1 = 1$$

The **fourth** term is:

$$\alpha_3 = \alpha_1 + \alpha_2 = 1 + 1 = 2$$

The **fifth** term is:

$$\alpha_4 = \alpha_2 + \alpha_3 = 1 + 2 = 3$$

The **sixth** term is:

$$\alpha_5 = \alpha_3 + \alpha_4 = 2 + 3 = 5$$

El **(n+1)th** term is:

$$\alpha_{n+1} = \alpha_{n-1} + \alpha_n$$

2. To calculate the **greatest common divider (GCD)** of two integers I can apply the **Euclid's** algorithm, which consists of **subtracting the smallest from the largest** until two equal **numbers remain**, which will be the maximum common divider of the two numbers. If we started with the pair of numbers 412 and 184, we would have:

| | | | | | | | | | | | | |
|-----|-----|-----|-----|----|----|----|----|----|----|----|---|---|
| 412 | 228 | 44 | 44 | 44 | 44 | 44 | 36 | 28 | 20 | 12 | 8 | 4 |
| 184 | 184 | 184 | 140 | 96 | 52 | 8 | 8 | 8 | 8 | 8 | 4 | 4 |

That is to say, **GCD 412, 184)=4**

