

UNIT 2. Introduction to Java programming

1	Introduction.....	4
2	Java fundamentals.....	5
2.1	Terminology	5
2.2	The java basic syntax	9
2.3	Java comments	11
2.4	Java Data Types	12
2.4.1	Java type conversion for primitive types	15
	Widening or Automatic Type Conversion	15
	Narrowing or Explicit Conversion	17
	Type promotion in Expressions	19
2.5	Java reserved keywords	20
2.6	Identifiers.....	20
2.7	Variables	21
	How variables are kept in RAM.....	21
2.8	Console input/output in Java	28
2.9	Java Operators.....	31
2.9.1	Assignment operators.....	32
2.9.2	Arithmetic Operators	34
2.9.3	Binary Operators / bitwise operators	35
	Practice	36
2.9.4	Java Booleans and boolean operators.....	37
2.9.5	Operators that return as a result booleans.....	37
	Java Comparison Operators.....	37
	Java Logical Operators	38
2.9.6	Control structures in Java	40
2.9.7	Introduction to Java Conditions and If Statements	43
	Activity. Cornell notes:.....	47
	Practice. Try it yourself	47
2.9.8	Characters and strings.....	47
3	Object-Oriented Language.....	51
	Pillars of Object-Oriented Programming.....	53
	Abstraction	54
	Encapsulation	55
	Inheritance	56
	Polymorphism	56
3.1	Java as an Object-Oriented programming	57
3.1.1	The this operator, reference to Object vs reference to class	61
3.2	Creating objects	62
	But, What happens when you create an object in java?.....	63
3.2.1	Public vs private properties. Encapsulation	64
	Methods. Encapsulation II. Private vs Public	66
	Accessor Methods.....	70
3.2.2	Static properties and methods.	74
3.3	Inheritance in Java	79

Java Programming	Carlos Cano Ladera
Protected vs private	82
Extending the inheritance hierarchy	83
3.4 Polymorphism	87
Overloading	87
Practice	89
Overriding	89
3.5 Abstract Classes.....	91
Practice	94
Following the last example, could you do the same for all classes?.....	94
3.6 Interfaces	94
Practice	100
3.7 The Object Class in Java	100
Practice	101
3.8 The java Object class	101
3.8.1 Equals and hashCode.....	105
3.8.2 Overriding hashCode and Equals in our classes	107
Practice	111
3.9 Enum types in Java	111
3.9.1 More complex enumerate types.....	114
Activity.....	118
4 Property, Variable, and parameter scope and passing parameters techniques.....	118
4.1 Variable scope	118
Practice	122
4.1.1 Passing Parameter techniques in Java	122
5 The java APIs.	130
5.1.1 Java.base, java.lang y javax	131
▪ Basic java libraries, expanding knowledge	131
5.2 Get started with the JAVA API.....	135
Built-in Packages	135
Practice	137
5.3 Importing our own classes.....	137
5.4 Wrap classes for primitive types. Type Conversion.....	137
6 Index	141
6.1 Java keywords.....	141

Color coding

Yellow: important definitions. / code statements of high relevance

Blue: concepts and topics / variable declarations

Red: key topics/ issues

1 Introduction

In this unit, you will start exploring all the fundamental features of the Java language. The chapter will try to first deconstruct Java, going over its elements or constituent which we learn in the previous unit. After that, we will plunge into the Java language itself. Syntax, operators, types, and oriented object features the from the author's perspective, there is no better way of learning than that use it, thus let's get to work on Java.

Java has emerged as the object-oriented programming language of choice. Some of the important concepts of Java include:

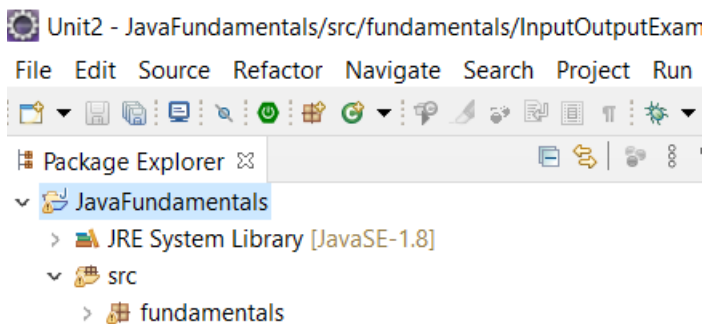
- A Java virtual machine (JVM), which provides the fundamental basis for the platform
- independence
- Automated storage management techniques, such as garbage collection
- Language syntax that is similar to that of the C language

The result is a language that is object-oriented and efficient for application programming. The chapter will try to present all the tools Java offers to write programs. This section covers the following topics:

- Key Components of the Java Language
- Object-Oriented java structure
- Java Object-Oriented Programming features

2 Java fundamentals

The first thing you must do to follow this chapter is to create a new workspace for unit2, Unit2/javaProjects. Secondly, create a new project called JavaFundamentals.



2.1 Terminology

To talk about Java fundamentals, we need to introduce some terminology. To do so, we start with the following code fragment:

```
int a, b, c;  
a = 1234;  
b = 99;  
c = a + b;
```

The first line is a **declaration statement** that declares the names of three variables using the identifiers a, b, and c and their type to be int.

The next three lines are **assignment statements** that change the values of the variables, using the **literals** 1234 and 99, and

Literals

A **literal** is a Java-code representation of a data-type value. We use sequences of digits such as 1234 or 99 to represent values of type int; we add a decimal point, as in 3.14159 or 2.71828, to represent values of type double; we use the keywords true or false to represent the two values of type boolean; and we use sequences of characters enclosed in matching quotes, such as "Hello, World", to represent values of type String.

Operators

An **operator** is a Java-code representation of a data-type operation. Java uses + and * to represent addition and multiplication for integers and floating-point numbers; Java uses &&, |, and ! to represent boolean operations; and so forth. We will describe the most commonly used operators on **built-in** (inherent; innate.) types later in this section.

Identifiers

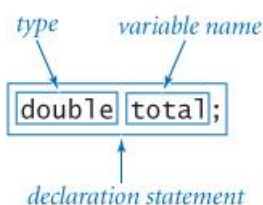
An **identifier** is a Java-code representation of a name (such as for a variable). Each identifier is a sequence of letters, digits, underscores, and currency symbols, the first of which is not a digit. For example, the sequences of characters `abc`, `Ab $`, `abc123`, and `a_b` are all legal Java identifiers, but `Ab*`, `1abc`, and `a + b` are not. Identifiers are case sensitive, so `Ab`, `ab`, and `AB` are all different names. Certain reserved words— such as `public`, `static`, `int`, `double`, `String`, `true`, `false`, and `null`— are special, and you cannot use them as identifiers.

Variables

A **variable** is an entity that holds a data-type value, which we can refer to by name. In Java, each variable has a specific type and stores one of the possible values from that type. For example, an `int` variable can store either the value `99` or `1234` but not `3.14159` or `"Hello, World"`. Different variables of the same type may store the same value. Also, as the name suggests, the value of a variable may change as a computation unfolds. For example, we use a variable named `sum` in several programs in this book to keep the running sum of a sequence of numbers. We create variables using declaration statements and compute with them in expressions, as described next.

Declaration statements

To create a variable in Java, you use a **declaration statement**, or just a **declaration** for short. A declaration includes a type followed by a variable name. Java reserves enough memory to store a data-type value of the specified type and associates the variable name with that area of memory so that it can access the value when you use the variable in later code. For economy, you can declare several variables of the same type in a single declaration statement.



Anatomy of a declaration

The text `"double total;"` is labeled as a **"declaration statement."** The text `"double"` is labeled `"type"` and the text `"total"`.

Constant variables

We use the oxymoronic term **constant variable** to describe a variable whose value does not change during the execution of a program (or from one execution of the program to the next). In this book, our convention is to give each constant variable a name that consists of an uppercase letter followed by uppercase letters, digits, and underscores. For example, we might use the constant variable names `SPEED_OF_LIGHT` and `DARK_RED`.

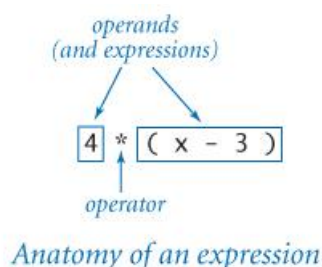
In Java, we use the `final` modifier to build constants variables. For name convention, programmers are accustomed to using capital letters and underscore “`_`” to separate multi-word constants given that they provided a way to distinguish variables constants from regular variables.

```
final int MAX_TEMP = 120;
```

Expressions

A **Java expression** is a combination of literals, variables, and operations that Java evaluates to produce a value. For primitive types, expressions often look just like mathematical formulas, using operators to specify data-type operations to be performed on one more operand. Most of the operators that we use are binary operators that take exactly two operands, such as `x - 3` or `5 * x`.

Each operand can be any expression, perhaps within parentheses. For example, we can write `4 * (x - 3)` or `5 * x - 6` and Java will understand what we mean. An expression is a directive to perform a sequence of operations; the expression is a representation of the resulting value.



We can distinguish between simple expressions, such as `x-3`, and complex expressions. Complex expressions are made up of simple expressions, operands, and parenthetical expressions. We use parenthesis to alter the **order of precedence**. For example, In Java, the product is evaluated before addition and subtraction. Providing that `x= 2`, `4*x-3` should be evaluated following this order:

1. `4*2-3= 8 -3`
2. `8-3=5`

Sometimes it is needed to change this order of precedence. To do so, we use the parenthetical expression like $4*(x-3)$. Regarding parentheses, the java compiler will examine first which is inside the parenthesis. As a result, this expression would evaluate in this order.

1. $4*(2-3) = 4*-1$
2. $4*-1 = -4$

Nested parenthetical expression

What if we have parenthetical expressions within parenthetical expressions? If you have studied math, you already know the answer. Because the java compiler, as almost any programming language compiler, follows the mathematical order of precedence, considering that inner parenthetical expressions are solved first.

Let's go over this complex expression $x * (x / (x + 4))$:

Supposing that $x=4$. This should be the natural evaluation in java:

Firstly, the most inner parenthetical expression $(x+4)$
 $4*(4/4+4) = (4*(4/8))$

Secondly, the other inner $(x/(x+4))$

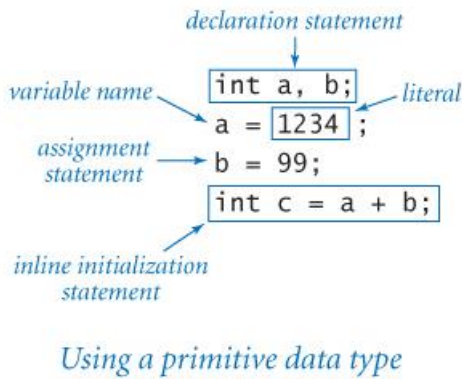
$4*(4/8) = 4*(0,5)$

And finally, the outer product $x * (x / (x + 4))$:

$4*(0,5) = 2$

Assignment statements

An **assignment statement** associates a data-type value with a variable. When we write $c = a + b$ in Java, we are not expressing mathematical equality, but we are expressing an action instead: set the value of the variable c to be the value of a plus the value of b . It is true that the value of c is mathematically equal to the value of $a + b$ immediately after the assignment statement has been executed, but the aim of the statement is to change (or initialize) the value of c . The left-hand side of an assignment statement must be a single variable; the right-hand side can be any expression that produces a value of a compatible type. So, for example, both $1234 = a$; and $a + b = b + a$; are invalid statements in Java. In short, the meaning of $=$ is decidedly not the same as in mathematical equations.



A section of the program statements read “int a, b; a = 1234 ; b = 99; int c = a + b;” The statement “int a, b;” is labeled “Declaration statement.” The letter ‘a’ in the statement “a = 1234 ;” is labeled as “Variable name” and the number “1234” is referred to as “Literal.” The statement “b = 99;” is called an “Assignment statement.” The statement “int c = a + b;” is named “Inline initialization statement.”

Inline initialization

Before you can use a variable in an expression, you must first declare the variable and assign to it an initial value. Failure to do either results in a compile-time error. For economy, you can combine a declaration statement with an assignment statement in a sort of sentence known as an **inline initialization statement**. For example, the following code declares two variables a and b, and initializes them to the values 1234 and 99, respectively:

```
int a = 1234;
```

2.2 The java basic syntax

Let us review the unit1 java example to briefly explain **Java syntax**. Predominantly, in Java, we declare three kinds of program statements. There will be more instructions unraveled in this course, but we will start with this three.

Here, we suggest adding the following code to your JavaFundamentals project.

```
package fundamentals;

public class MainClass {

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

```

        System.out.println("Hello there. Howdy. My first java program on the
running");

    }

}

```

In java, we **declare classes**, blueprints for objects:

```
public class MainClass {
```

We use the preserved word **class**. If the class is the main class in the file, you should add the **public** modifier first. As a reminder, the name of the java file must be the same as the public class name. There can be only one public class per file. Hence, the file name for this class must be MainClass.java.

Moreover, **we declare methods**, similar to subroutines. They have method signature and body.

```

static void main(String[] args) → SIGNATURE

{
    // TODO Auto-generated method stub

    System.out.println("Hello there. Howdy.    → BODY
My first java program on the running");
}

```

The function signature has the following format:

Modifiers --- Return value ---- method_Name ---- parameters.
static **void** main (String[] args)

1. Modifiers: add extra meaning to java structures, such as methods or classes.
2. Return value: void if the method returns nothing. A data type such as int or String, in all other cases.
3. Function_name: we will use this to call the method.
4. Parameters: values received by the function. They are separated by commas, and they are placed in between parenthesis.

The body is a block of instructions or code. It starts and ends with curly braces {}.

```

{
    // TODO Auto-generated method stub

```

```
System.out.println("Hello there. Howdy. My first java program on the running");  
}
```

→ BODY

Developers write instructions or sentences in Java. There are two types of sentences.

1. Single line statement. They contain function calls, operators, and literals (plain data).

```
int i = i+5;
```

or

```
System.out.println("Hello there. Howdy. My first java program on the running");
```

2. Block statement. A sentence that contains a block of code. Curly braces outline the beginning and end of a block of code. Notice that the statements inside the block statement are indented. The Eclipse IDE uses tabulation to indent our code. It makes the code look organized and it is easier to identify which statements are within a block statement.

```
while (i<100) {  
    i = i+5;  
    cont = cont+1;  
}
```

2.3 Java comments

Comments can be used to explain Java code, and to make it more readable. It can also be used to prevent execution when testing alternative code.

Single-line comments start with two forward slashes (//).

Any text after the double-slash “//” and the end of the line is ignored by Java (will not be executed).

This example uses a single-line comment before a line of code:

```
// This is a comment
```

```
System.out.println("Hello World");
```

Java Multi-line Comments

Multi-line comments start with `/*` and ends with `*/`.

Any text between `/*` and `*/` will be ignored by Java.

This example uses a multi-line comment (a comment block) to explain the code:

Example

```
/* The code below will print the words Hello World  
to the screen, and it is amazing */  
System.out.println("Hello World");
```

2.4 Java Data Types

As explained for Flowgorithm flowcharts, a variable in Java must have a specific data type.

Data types are divided into two groups:

- Primitive data types - includes `byte`, `short`, `int`, `long`, `float`, `double`, `boolean` and `char`. Represent the basic data we can store in variables. Numbers, 9, 7,6, characters such as for `'?'`.
- Non-primitive data types - such as `String`, `Arrays` and `Classes` (you will learn more about these later in the chapter)

Example

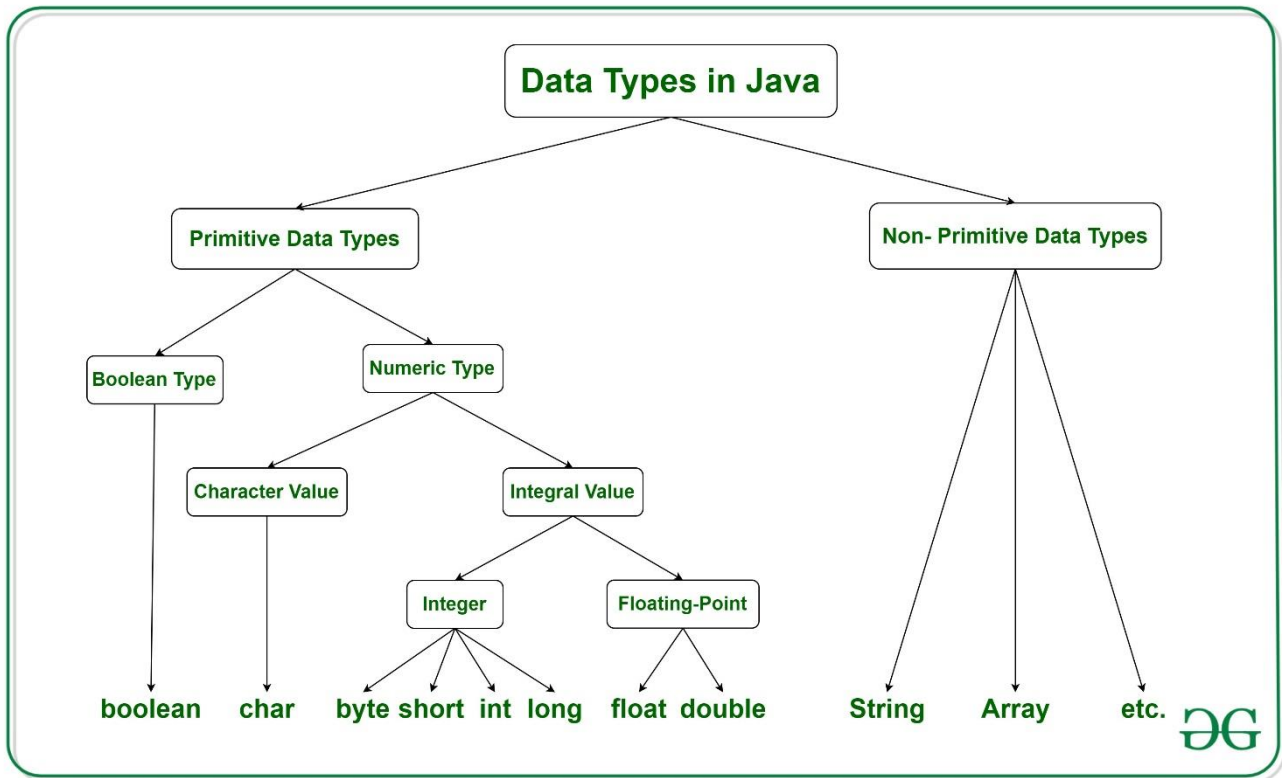
```
int myNum = 5;           // Integer (whole number)  
float myFloatNum = 5.99f; // Floating point number  
char myLetter = 'D';     // Character  
boolean myBool = true;   // Boolean  
String myText = "Hello"; // String
```

- **byte:** The `byte` data type is an 8-bit signed two's complement integer. It has a minimum value of -128 and a maximum value of 127 (inclusive). The `byte` data type can be useful for saving memory in large arrays, where the memory savings actually matters. They can also be used in place of `int` where their limits help to clarify your code; the fact that a variable's range is limited can serve as a form of documentation.
- **short:** The `short` data type is a 16-bit signed two's complement integer. It has a minimum value of -32,768 and a maximum value of 32,767 (inclusive). As with `byte`, the same guidelines apply: you can use a `short` to save memory in large arrays, in situations where the memory savings actually matters.
- **int:** By default, the `int` data type is a 32-bit signed two's complement integer, which has a minimum value of -2^{31} and a maximum value of $2^{31}-1$. In Java SE 8 and later, you can use the `int` data type to represent an unsigned 32-bit integer, which has a minimum value of 0 and a maximum value of $2^{32}-1$. Use the `Integer` class to use the `int` data type as an unsigned integer. See the section [The Number Classes](#) for more information. Static methods like `compareUnsigned`, `divideUnsigned`, etc have been added to the `Integer` class to support the arithmetic operations for unsigned integers.
- **long:** The `long` data type is a 64-bit two's complement integer. The signed long has a minimum value of -2^{63} and a maximum value of $2^{63}-1$. In Java SE 8 and later, you can use the `long` data type to represent an unsigned 64-bit long, which has a minimum value of 0 and a maximum value of $2^{64}-1$. Use this data type when you need a range of values wider than those provided by `int`. The `Long` class also contains methods like `compareUnsigned`, `divideUnsigned` etc to support arithmetic operations for unsigned long.
- **float:** The `float` data type is a single-precision 32-bit IEEE 754 floating point. Its range of values is beyond the scope of this discussion, but is specified in the [Floating-Point Types, Formats, and Values](#) section of the Java Language Specification. As with the recommendations for `byte` and `short`, use a `float` (instead of `double`) if you need to save memory in large arrays of floating point numbers. This data type should never be used for precise values, such as currency. For that, you will need to use `java.math.BigDecimal` class instead. [Numbers and Strings](#) cover `Decimal` and other useful classes provided by the Java platform.
- **double:** The `double` data type is a double-precision 64-bit IEEE 754 floating point. Its range of values is beyond the scope of this discussion, but is specified in the [Floating-Point Types, Formats, and Values](#) section of the Java Language Specification. For decimal values, this data type is generally the default choice. As mentioned above, this data type should never be used for precise values, such as currency.
- **boolean:** The `boolean` data type has only two possible values: `true` and `false`. Use this data type for simple flags that track true/false conditions. This data type represents one bit of information, but its "size" isn't something that's precisely defined.
- **char:** The `char` data type is a single 16-bit Unicode character. It has a minimum value of `'\u0000'` (or 0) and a maximum value of `'\uffff'` (or 65,535 inclusive).

Type	Data	Size	range
------	------	------	-------

Byte	numeric	8 bits	-128-127
short	numeric	16 bits	
int	numeric	32 bits	
long	numeric	64 bits	
float	numeric	32 bits	
double	numeric	64 bits	
boolean	Logic		
char	character	16 bits	

Java is **statically typed** and a **strongly typed language** because, in Java, each type of data (such as integer, character, hexadecimal, packed decimal, and so forth) is predefined as part of the programming language and all constants or variables are defined for a given program must be constrained with one of the data types.



2.4.1 Java type conversion for primitive types

When you assign any data type value to another typed variable, the two types might not be compatible with each other. If the data types are compatible, then Java would perform the conversion automatically known as **Automatic Type Conversion**, and if not then they need to be cast or converted explicitly. For example, assigning an int value to a long variable.

Widening or Automatic Type Conversion

Widening conversion takes place when two data types are automatically converted. This happens when:

- The two data types are compatible.
- When we assign the value of a smaller data type to a bigger data type.

For Example, in Java, the numeric data types are compatible with each other but no automatic conversion is supported from numeric type to char or boolean. Also, char and boolean are not compatible with each other.

Byte → Short → Int → Long → Float → Double

Widening or Automatic Conversion

JavaTypeConversion.java

In the following code, this program declares variables from any numeric type in java. It initially starts with the shorter type 8 bytes. Then assigning to a bigger type of variable, it automatically converts the “100” number to integer or decimal types. Since a byte is the smaller type, the code never loses information in the automatic conversion.

```
package fundamentals;

public class JavaTypeConversion {

    public static void main(String[] args)

    {
        byte b= 100;
        short s= b;

        int i = s;

        // automatic type conversion
        long l = i;

        // automatic type conversion
        float f = l;

        double d = b;

        System.out.println("Byte value "+b);
        System.out.println("Short value "+s);
        System.out.println("Int value "+i);
        System.out.println("Long value "+l);
        System.out.println("Float value "+f);
        System.out.println("double value "+f);
    }

}
```


Execution example

```
Byte value 100  
Short value 100  
Int value 100  
Long value 100  
Float value 100.0
```

```
double value 100.0
```

Narrowing or Explicit Conversion

If we want to assign a value of a larger data type to a smaller data type, we perform explicit type **casting or narrowing**.

This is useful for incompatible data types where automatic conversion cannot be done. Here, the target type specifies the desired type to convert the specified value to.

Note: Char and number are not compatible with each other. Let's see when we try to convert one into other

Double → Float → Long → Int → Short → Byte

Narrowing or Explicit Conversion

Notice how your Eclipse IDE will display a compile error in the next example

ExplicitConversionExampleError.java

```
package fundamentals;  
  
public class ExplicitConversionExampleError {  
    public static void main(String[] argv)  
    {  
        char ch = 'c';  
        int num = 88;  
  
        double numDec=44.5;  
        ch = num;  
        ch = numDec;
```

```
}
```

```
}
```

Error:

7: error: incompatible types: possible lossy conversion from int to char

```
    ch = num;
```

```
        ^
```


1 error

How to do Explicit Conversion?

To fix this error, Java provides a tool to implement Explicit conversion-which is labeled as “**Casting**”. Casting syntax consists of writing down the type between parentheses on the right side of the assignment statement, just before the variable or expression.

Casting

Variable = (Type) expression;



```
b = (byte) i;
```

```
b = (byte) i + 5
```

While value to a byte type is assigned, the fractional part is lost and is reduced to modulo 256(range of byte).

ExampleConversionTypeLoss.java

```
package fundamentals;

public class ExampleConversionTypeLoss {

    public static void main(String args[])
    {
        byte b;
        int i = 257;
        double d = 323.142;
```

```

        System.out.println("Conversion of int to byte.");

        //i%256
        b = (byte) i;
        System.out.println("i = " + i + " b = " + b);
        System.out.println("\nConversion of double to byte.");

        //d%256
        b = (byte) d;
        System.out.println("d = " + d + " b= " + b);
    }

}

```

Conversion of int to byte.
i = 257 b = 1

Conversion of double to byte.
d = 323.142 b= 67

Type promotion in Expressions

While evaluating expressions, the intermediate value may exceed the range of operands, and thus the expression value will be promoted. Some conditions for type promotion are:

3. Java automatically promotes each byte, short, or char operand to int when evaluating an expression.
4. If one operand is a long, float, or double the whole expression is promoted to long, float, or double respectively.

ExplicitConversionExample.java

```

package fundamentals;

public class ExplicitConversionExample {

    public static void main(String[] args)
    {
        double d = 100.04;

        //explicit type casting
        long l = (long)d;

        //explicit type casting
    }
}

```

```

    int i = (int)l;
    System.out.println("Double value "+d);

    //fractional part lost
    System.out.println("Long value "+l);

    //fractional part lost
    System.out.println("Int value "+i);
}

}

```

Execution

```

Double value 100.04
Long value 100
Int value 100

```

2.5 Java reserved keywords

Here is a list of keywords in the Java programming language. You cannot use any of the following as identifiers in your programs. The keywords `const` and `goto` are reserved, even though they are not currently used. `true`, `false`, and `null` might seem like keywords, but they are actually literals; you cannot use them as identifiers in your programs.

<code>abstract</code>	<code>continue</code>	<code>for</code>	<code>new</code>	<code>switch</code>
<code>assert***</code>	<code>default</code>	<code>goto*</code>	<code>package</code>	<code>synchronized</code>
<code>boolean</code>	<code>do</code>	<code>if</code>	<code>private</code>	<code>this</code>
<code>break</code>	<code>double</code>	<code>implements</code>	<code>protected</code>	<code>throw</code>
<code>byte</code>	<code>else</code>	<code>import</code>	<code>public</code>	<code>throws</code>
<code>case</code>	<code>enum****</code>	<code>instanceof</code>	<code>return</code>	<code>transient</code>
<code>catch</code>	<code>extends</code>	<code>int</code>	<code>short</code>	<code>try</code>
<code>char</code>	<code>final</code>	<code>interface</code>	<code>static</code>	<code>void</code>
<code>class</code>	<code>finally</code>	<code>long</code>	<code>strictfp**</code>	<code>volatile</code>
<code>const*</code>	<code>float</code>	<code>native</code>	<code>super</code>	<code>while</code>

2.6 Identifiers

An identifier is a Java-code representation of a name (such as for a variable). Each identifier is a sequence of letters, digits, underscores (`_`), and currency symbols (`$`), the first of which is not a digit. For example, the sequences of characters `abc`, `Ab $`, `abc123`, and `a_b` are all legal Java identifiers, but `Ab*`, `1abc`, and `a + b` are not. Identifiers are case sensitive, so `Ab`, `ab`, and `AB` are all different names. Certain reserved words— such as `public`, `static`, `int`, `double`, `String`, `true`, `false`, and `null`— are special, and you cannot use them as identifiers.

For instance:

\$1, ab\$3, _AB are valid identifiers

1fd 2\$7 would produce a compilation error -> you cannot start an identifier with a digit

2.7 Variables

Variables are containers for storing data values. To declare a variable, we state the type following the name of the variable.

Type name
int myNum

Data Type	name	Assign a value is optional
int	myNum	= 5;

```
package fundamentals;
```

```
public class VariableExamples {
```

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

```
    {
```

```
        int myNum = 5;
```

```
        int yourNum=myNum + 7;// Integer (whole number)
```

```
        float myFloatNum = 5.99f;
```

```
        double myDouble=0.0d;
```

```
        boolean respuesta=true;
```

```
        char capitalLetter = 'c';
```

```
        respuesta = false;
```

```
        myDouble= myFloatNum + myNum;
```

```
    }
```

How variables are kept in RAM memory

Our code, and variables included stored in memory. Therefore, each variable is bounded to a memory address or position.

RAM Memory

Variable naming conventions

Programmers typically follow stylistic conventions when naming things. In this course, our convention is to give each variable a meaningful name that consists of a lowercase letter followed by lowercase letters, uppercase letters, and digits. We use uppercase letters to mark the words of a multi-word variable name. For example, we use the variable names `i`, `x`, `y`, `sum`, `isLeapYear`, and `outDegrees`, among many others. Programmers refer to this naming style as camel case.

Memory address		Value In binary memory
..		
..		
..		
1000000	<u>myNum</u>	5
1000001	<u>yourNum</u>	12
1000002	<u>myFloatNum</u>	5.99
1000003	<u>myDouble</u>	0.0
1000004	<u>respuesta</u>	
1000005	..	
1000006	<u>respuesta=true</u>	
1000007	..	
1000008	<u>myDouble=</u> <u>myFloatNum</u> + <u>myNum;</u>	
1000009	..	

Java program

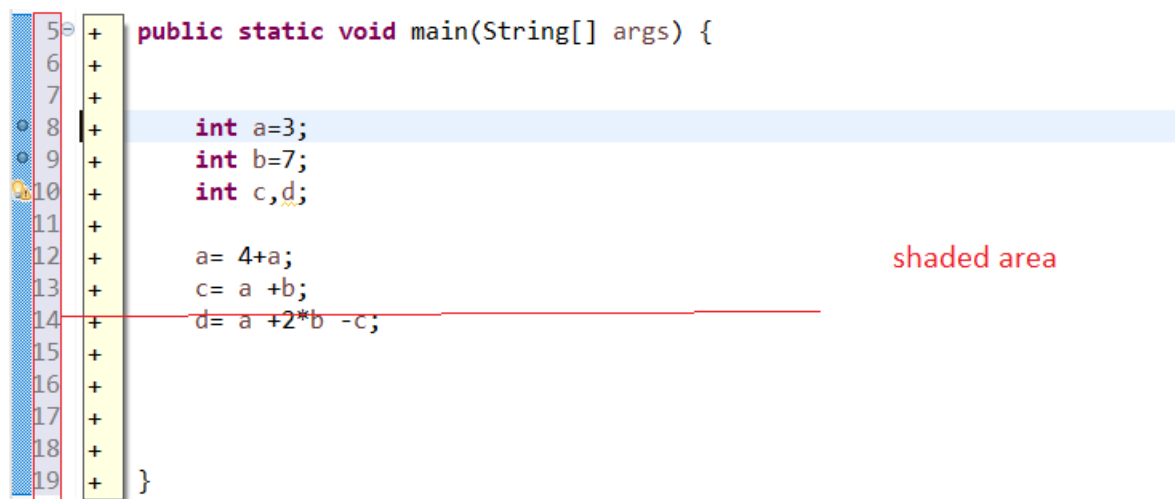
Tracing changes in variable values

As a final evaluation of your understanding of the purpose of assignment statements, convince yourself that the following code exchanges the values of a and b (assume that a and b are int variables):

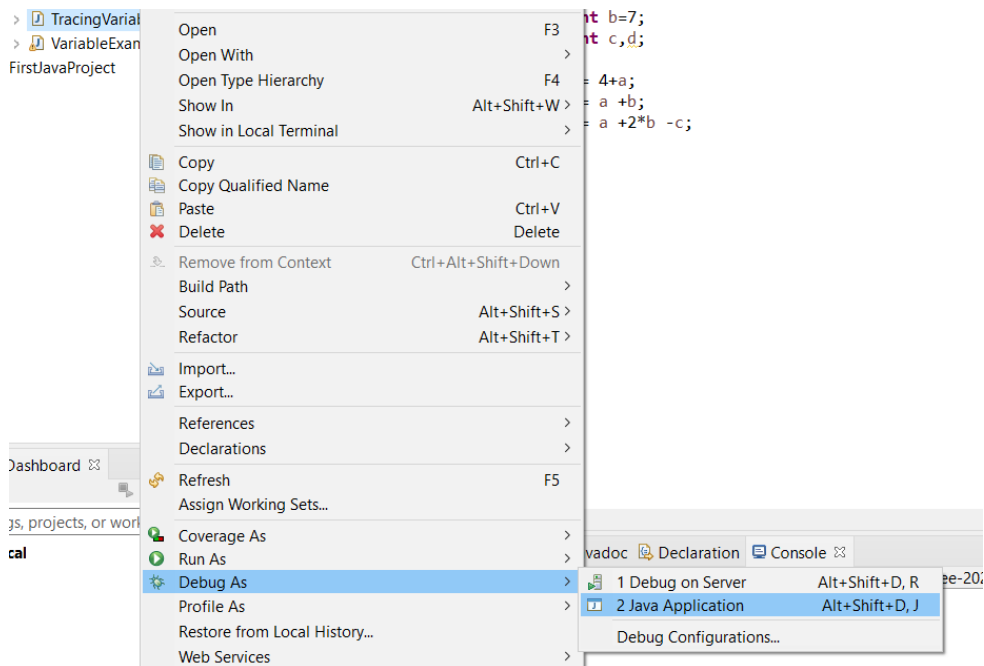
```
int t = a;  
a = b;  
b = t;
```

To do so, we introduce breakpoints to our program. Breakpoints are a tool offered by IDEs to halt your program executions at the given point, which is extremely useful, thus we can inspect variables values as well.

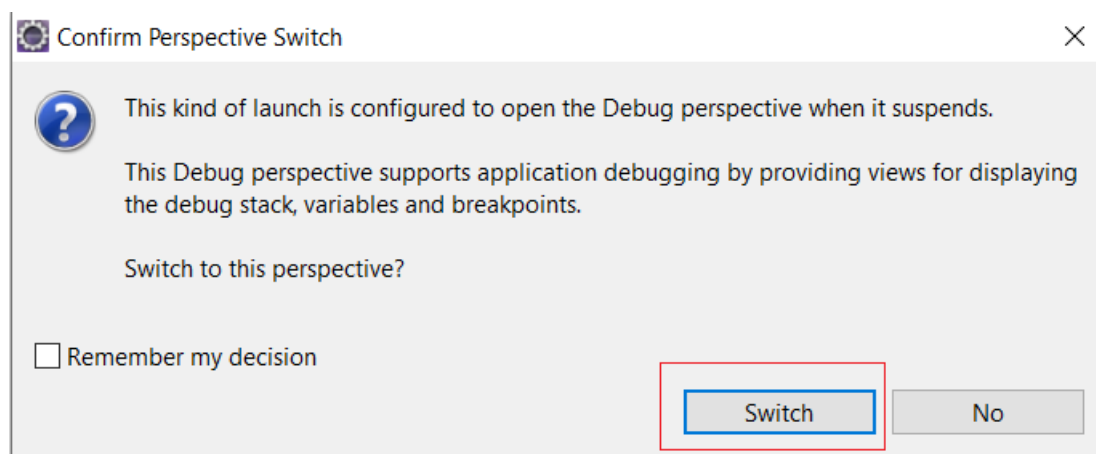
To add a breakpoint you can double click on the shaded area to the start of the IDE Editor Window. You should do in the baseline of the statement you want to pause your program in. As soon as you double-click a blue circle should appear left to the statement line number 8.

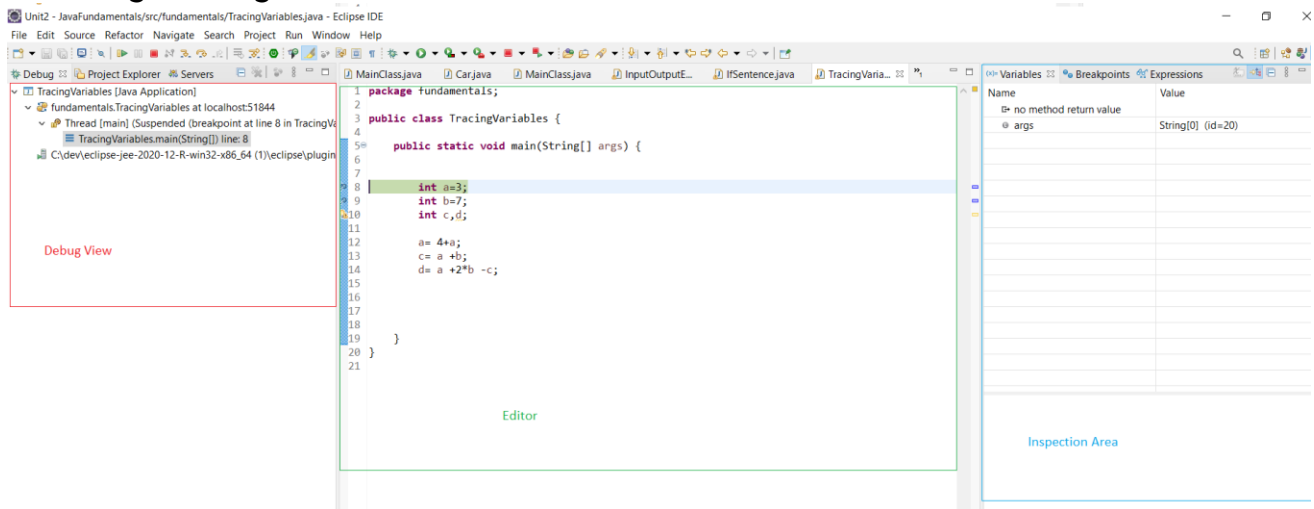


After fixing the breakpoint, to pause your program you should debug. Although it is a similar process to running a program, it allows developers to halt their programs, inspect store values in variables, and follow the program flow.

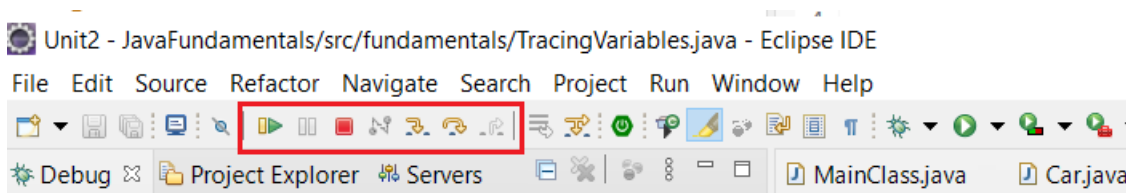


As long as you start debugging, the IDE requests to open the Debug perspective. In this perspective, you will be allowed to inspect variables and expressions. Let's do it.





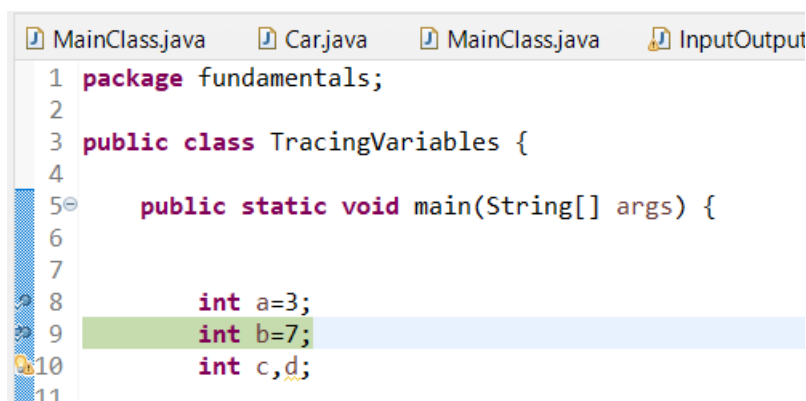
You may realize that the program has paused just in the first breakpoint line. Up to now, we have not declared yet any variable. Therefore, there is no other variable in the inspection area but the main parameters. To make your program progress we will use some toolbar debugging icons related to the Debug Perspective.



- The play button resumes the program
- The pause button pauses the program given that it is running
- The stop button, fully stop the program execution
- The Step into button, gets into a function, providing that the statement is a function call.
- The Step Over button, executes the statement and moves to the next one.

So, press the Step Over function and check the Inspect Area, looking for changes.

The program flow steps to the next line:



Because the “a” variable is declared, now we are allowed to check its value in the inspected area. The inline declaration `a=3`; produces the desired result.

Name	Value
↳ no method return value	
args	String[0] (id=20)
a	3

In memory a is storing now a 3 as a value. All the changes we are inspecting as we run the program are stored in memory, remember them.

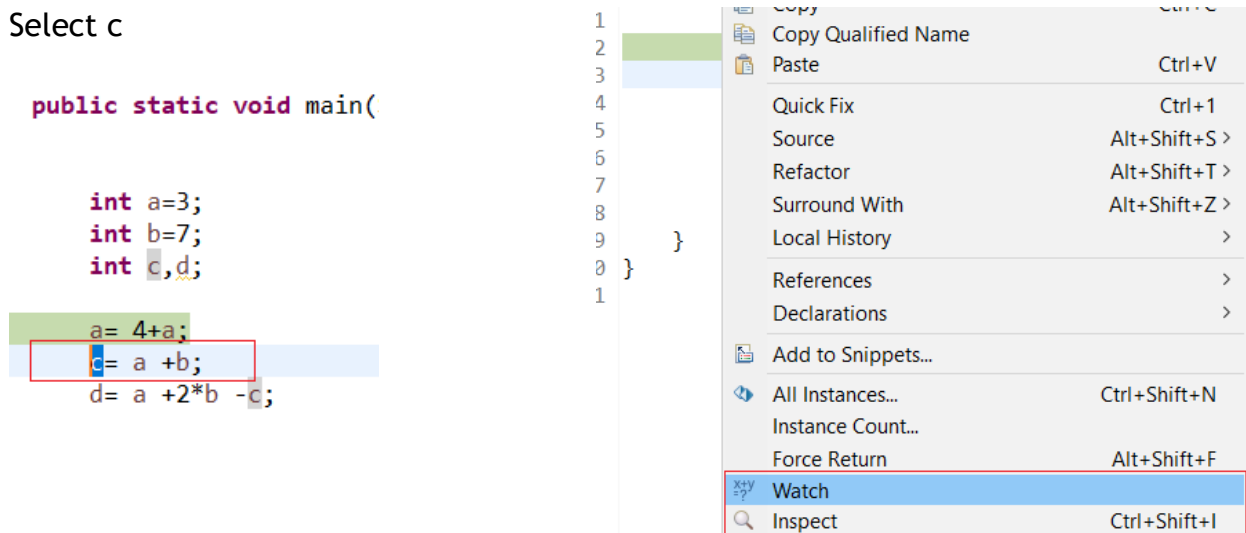
Keep executing just to line 12

```
1 package fundamentals;
2
3 public class TracingVariables {
4
5     public static void main(String[] args) {
6
7
8         int a=3;
9         int b=7;
10        int c,d;
11
12        a= 4+a;
13        c= a +b;
14        d= a +2*b -c;
15
16
17
18
```

Name	Value
↳ no method return value	
args	String[0] (id=20)
a	3
b	7

Variables a and b are stored values, yet c and d are not. Even if variables are declared, they do not have values assigned yet; Their value is undefined. We will try to watch or inspect the c and d variables. To perform so, you can select the c variable, the mouse right button click, and click on the watch or inspect option in the pop-up menu.

Select c

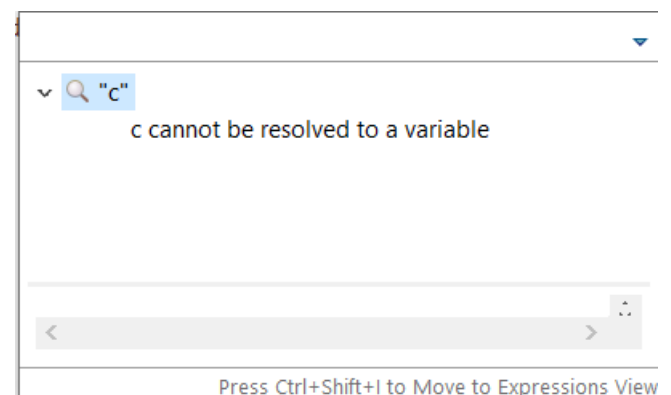


In the inspection are in expressions, c is added. Owing to that it does not store a value, there is an evaluation error. C cannot be resolved to a variable. The variable value is undefined.

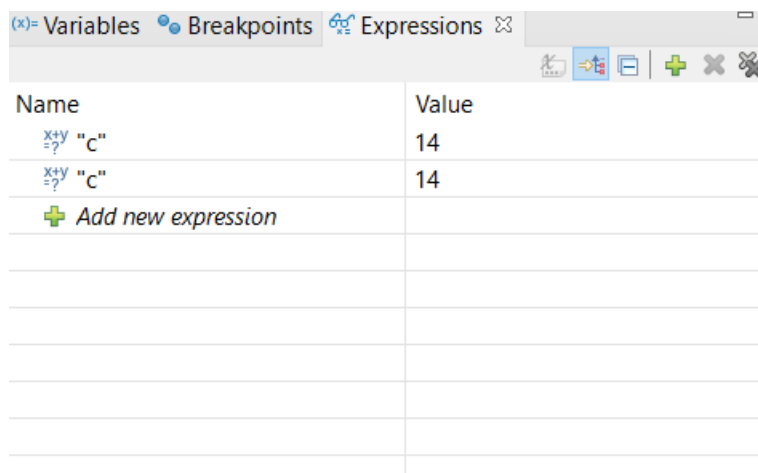
Watch

Name	Value
✓ "c"	<error(s)_during_the_evaluation:>
c cannot be resolved to a va	
> "c"	<error(s)_during_the_evaluation:>
+ Add new expression	

Inspect



As you move forward to the 14th line, the `c` variable has already given a value. The variable is not only declared, but also defined.



The screenshot shows a window titled '(x)=' with tabs for 'Variables', 'Breakpoints', and 'Expressions'. The 'Variables' tab is active, displaying a table with two columns: 'Name' and 'Value'. The table contains two entries for the variable 'c', both with a value of 14. Below the table is a button labeled '+ Add new expression'.

Name	Value
<code>x+y</code> <code>"c"</code>	14
<code>x+y</code> <code>"c"</code>	14
+ Add new expression	

All in all, programmers use IDE's debugging tools to verify:

1. The program flow.
2. Variable values
3. The program does what it is intended to
4. Searching for programming errors.

2.8 Console input/output in Java

For console input, Java offers the `Scanner` class. Later we will introduce the concept of classes. From now on you should use the `Scanner` class as explained.

For output we use the `System.out` library methods, `System.out.println()`, `System.out.printf()`, `System.out.print()`. It will be explained by examples. So, let's have a look at the next example `InputOutputExample.java`:

```
package fundamentals;
```

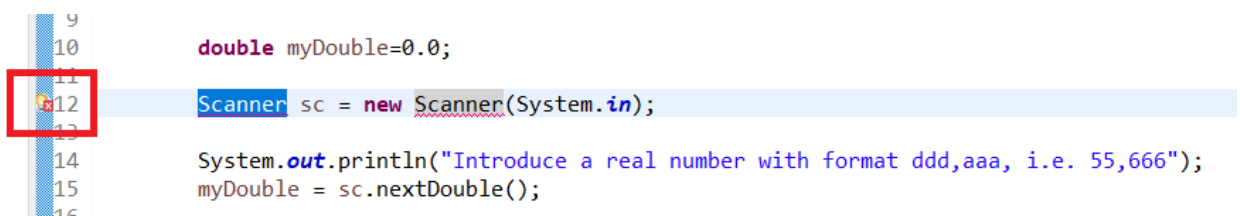
```
import java.util.Scanner;
```

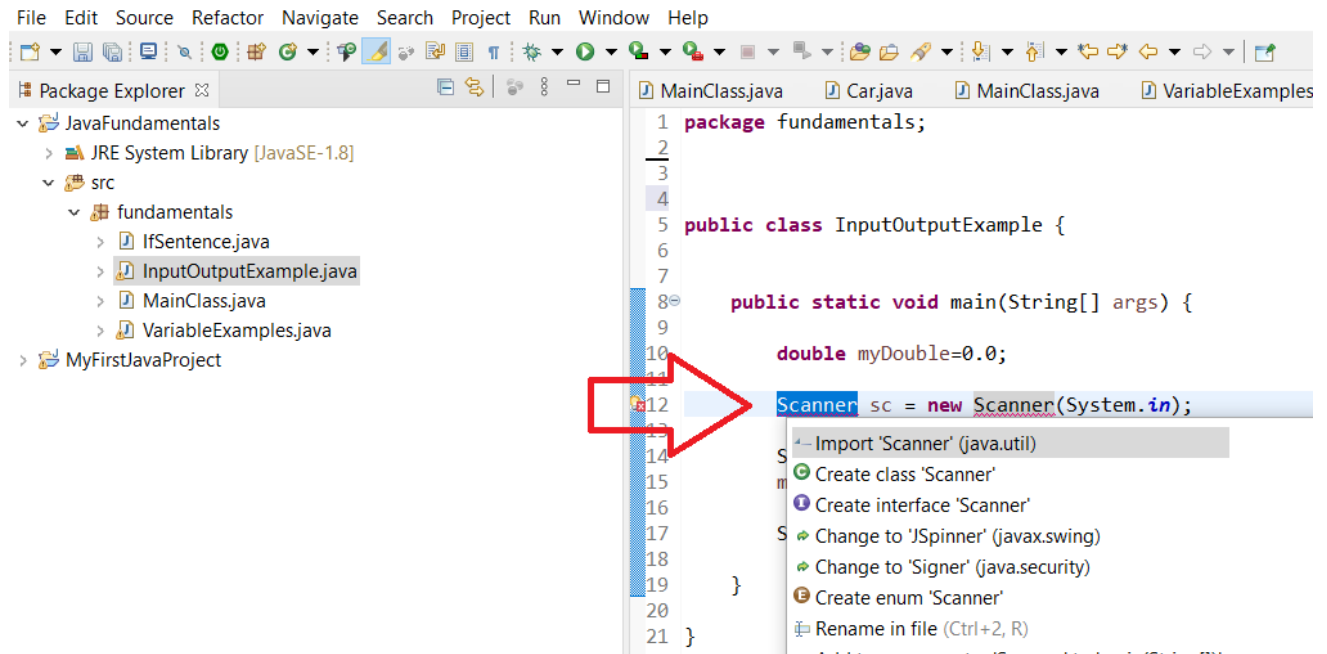
```
public class InputOutputExample {  
  
    public static void main(String[] args) {  
  
        double myDouble=0.0;  
  
        Scanner sc = new Scanner(System.in);  
  
        System.out.println("Introduce a real number with format ddd.aaa, i.e.  
55.666");  
        myDouble = sc.nextDouble();  
  
        System.out.println("The number readed from the input is:"+ myDouble);  
  
    }  
}
```

Java allows programmers to develop Java program offering the Java API. Some of the classes are added implicitly to programs. Some others need to be imported from the Java API libraries. We will explain the API concept in detail in this chapter. However, just to start, we will need to add some imports to our programs.

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

The Eclipse Ide tells you when to do it. In the event that the previous import is not included in your program, Eclipse will show a compilation error. As a consequence, you would find the compilation error since the sentence or some of the words of your sentence are underlined in red. The Ide would suggest how to fix the problem. To obtain the messages, click on the red x icon to the left of the sentence.





We create a Scanner object, and we associate it with the standard input in java `System.in`

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

```
myDouble = sc.nextDouble();
```

will obtain the next double from the console.

For the output `println` write Strings in the Java standard output, the console.

```
System.out.println("The number read it from the input is:" + myDouble);
```

Now, you can execute the program and check for results.

Introduce a real number with format ddd,aaa, i.e. 55,666

55,9

The number read it from the input is:55.9

Data types were described in the subchapter above. Scanner provides methods to get data from the console for all Java primitive type. Some of the methods are listed below:

Method	Description
<code>nextInt()</code>	reads an <code>int</code> value from the user
<code>nextFloat()</code>	reads a <code>float</code> value form the user
<code>nextBoolean()</code>	reads a <code>boolean</code> value from the user
<code>nextLine()</code>	reads a line of text from the user (String)
<code>next()</code>	reads a word from the user
<code>nextByte()</code>	reads a <code>byte</code> value from the user
<code>nextDouble()</code>	reads a <code>double</code> value from the user
<code>nextShort()</code>	reads a <code>short</code> value from the user
<code>nextLong()</code>	reads a <code>long</code> value from the user

2.9 Java Operators

This section will provide the complete list of java operators. In program instructions, we combine operators, literals, and variables to perform operations in each of our program sentences. Here is an example.

```
int x= 0; // the value of x is 0
int y; // the value of y is undefined yet
```

```
x=x+5; // the value of x is 5
x++; // the value of x is 6
```

```
y=3+x; // the value of y is 9
```

To try out operators you should visit the w3c website.

https://www.w3schools.com/java/java_operators.asp

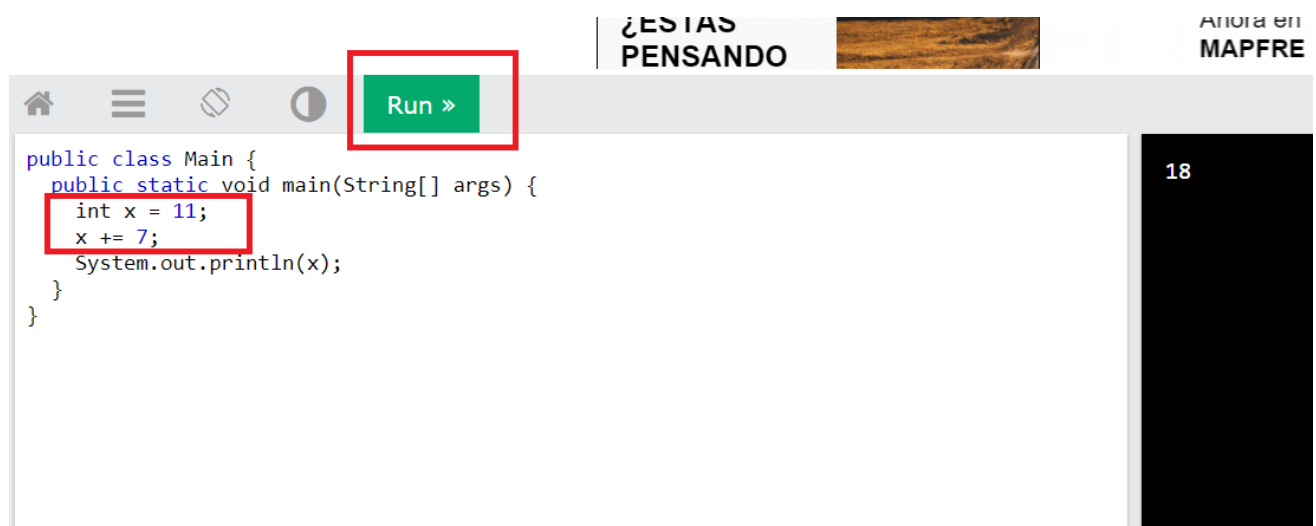
Click on the try yourself button

Example

```
int x = 10;  
x += 5;
```

[Try it Yourself »](#)

Change variables values, and run the program by pressing the run button



2.9.1 Assignment operators

Assignment operators are used to assigning values to variables. In the example below, we use the assignment operator (=) to assign the value 10 to a variable called x:

```
int x = 10;
```

The addition assignment operator (+=) adds a value to a variable:

Example

```
int x = 10;
```

```
x += 5; // it is the same as x= x+5; x contains 15 after the execution
```


A list of all assignment operators:

Operator	Example	Same As	Meaning
=	x = 5	x = 5	assignment
+=	x += 3	x = x + 3	Assignment + addition
-=	x -= 3	x = x - 3	Assignment + subtraction
*=	x *= 3	x = x * 3	Assignment + product
/=	x /= 3	x = x / 3	Assignment + division
%=	x %= 3	x = x % 3	Assignment + division remainder
&=	x &= 3	x = x & 3	Assignment + bitwise and

<code> =</code>	<code>x = 3</code>	<code>x = x 3</code>	Assignment + bitwise or
<code>^=</code>	<code>x ^= 3</code>	<code>x = x ^ 3</code>	Assignment + power $X = X^3$
<code>>>=</code>	<code>x >>= 3</code>	<code>x = x >> 3</code>	Assignment + bitwise left offset
<code><<=</code>	<code>x <<= 3</code>	<code>x = x << 3</code>	Assignment + bitwise right offset

For each type of operator, we will try out some of the W3C tutorial operators. Let's do it.

2.9.2 Arithmetic Operators

Arithmetic operator's usage is to perform common mathematical operations.

Operator	Name	Description	Example
<code>+</code>	Addition	Adds together two values	<code>x + y</code>
<code>-</code>	Subtraction	Subtracts one value from another	<code>x - y</code>

*	Multiplication	Multiplies two values	x * y
/	Division	Divides one value by another	x / y
%	Modulus	Returns the division remainder	x % y
++	Increment	Increases the value of a variable by 1	++x
--	Decrement	Decreases the value of a variable by 1	--x

2.9.3 Binary Operators / bitwise operators

Java defines several bitwise operators, which can be applied to the integer types, long, int, short, char, and byte. The Bitwise operator works on bits and performs the bit-by-bit operation. Assume if a = 60 and b = 13; now in the binary format they will be as follows –

```

a = 0011 1100
b = 0000 1101
-----
a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

```

The following table lists the bitwise operators and offers examples of use. In the mentioned example, assume integer variable A holds 60 and variable B holds 13 then –

Operator	Description	Example
& (bitwise and)	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
(bitwise or)	Binary OR Operator copies a bit if it exists in either operand.	(A B) will give 61 which is 0011 1101
^ (bitwise XOR)	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~ (bitwise compliment)	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<< (left shift)	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>> (right shift)	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 1111
>>> (zero fill right shift)	Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.	A >>>2 will give 15 which is 0000 1111

Practice

- 1) Create a class in your project called JavaAddNumbers.java. Also, add the main function and program this pseudocode in Java

```

PROGRAM JavaAddNumbers
    Declare Integer CAN1, CAN2, SUM

    Assign CAN1 = 5
    Assign CAN2 = 3
    Assign SUM = CAN1 + CAN2
    Output "THE SUM IS:"
    Output ToString(SUM)
End

```

- 2) Create a java program in a java class named AreaPentagon.java. It receives as an input the pentagon side length, and it prints the Regular Pentagon Area in the console. You should find the formula on the Internet.

2.9.4 Java Booleans and boolean operators

Very often, in programming, you will need a data type that can only have one of two values, like:

- YES / NO
- ON / OFF
- TRUE / FALSE

For this, Java has a `boolean` data type, which can take the values `true` or `false`.

Boolean Values

A boolean type is declared with the `boolean` keyword and can only take the values `true` or `false`:

Example

```
boolean isJavaFun = true;
boolean isFishTasty = false;
System.out.println(isJavaFun);    // Outputs true
System.out.println(isFishTasty);  // Outputs false
```

2.9.5 Operators that return as a result boolean types

Java Comparison Operators

Comparison operators are used to compare two values:

Operator	Name	Example	Try it
----------	------	---------	--------

==	Equal to	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

You should try some of the examples from the W3C tutorial in the link below:

https://www.w3schools.com/java/java_operators.asp

Java Logical Operators

Logical operators are utilized to determine the logic between variables or values:

Operator	Name	Description	Example
&&	Logical and	Returns true if both statements are true	x < 5 && x < 10

	Logical or	Returns true if one of the statements is true	<code>x < 5 x < 4</code>
!	Logical not	Reverse the result, returns false if the result is true	<code>!(x < 5 && x < 10)</code>

We will study in detail the logical operators going over their true chart.

And operator &&, true chart

Operation	Return value
<code>true && true</code>	<code>true</code>
<code>true && false</code>	<code>false</code>
<code>false && true</code>	<code>false</code>
<code>false && false</code>	<code>false</code>

The && operator will return true as a value if and only if both conditions are true. For example, let's say you have a variable `int x= 5` in your program

The logical operation `x>0 && x<10` should return true, because 5 is greater than 0 less than 10. If one of the conditions is false, the “and” && operator will return false. For `x>6 && x<10` would be false. Although 5 is less than 10, 5 is not greater than 6. In the end, both conditions must be true, to allow an and operation be true.

Or operator ||, true chart

Operation	Return value
<code>true true</code>	<code>true</code>
<code>true false</code>	<code>true</code>
<code>false true</code>	<code>true</code>
<code>false false</code>	<code>false</code>

The or operator || will return true a value provided that at least one of the conditions is true. Hence, supposing that you have a y variable `y=7`, the operation `y>8 || y<10` should be

true due to the fact that 7 is less than 10. Moreover, the `y>5 || y<10` should be true because both conditions are satisfied. `7>5` and `7<10` are both true.

The unary operator Not ! true chart

Operation	Return value
<code>!true</code>	<code>false</code>
<code>!false</code>	<code>true</code>

The not operator works by inverting (or negating) the value of its operand.. For instance, consider that you have a boolean variable `boolean bClose= true`. If you apply the NOT operator to this condition! `bClose` the result should be false, since the not operator returns the opposite of true, false.

Just to end this subchapter, try some of the examples from W3C for logical operators.

2.9.6 Control structures in Java

As we mentioned previously, in this section we will explain in detail the control flow structures we already introduced in Unit 1 for Flowcharts. This time we will 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 + assignment

```
int i=0;
```

2. Sequential statements

a. Assignment and expressions

i. Simple assignment

```
i=0;
```

ii. Complex assignment: 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 curly braces {}

```
{
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 reaching the end of the statement block.

5. Conditional or selection statements: the control of the flow of our program relying on logical conditions.

```
if (age >= 18)
    System.out.println("The person is an adult");
```

Iterative Statements: they execute a statement or block statement depending on conditions set up by the programmer. In the next excerpt of code, the while iterative statement would repeat 5 times, fulfilling the condition (`i < 5`). You can infer from the next example that iterative statements can contain block statements. Besides, conditional statements can contain block statements.

```
int i = 0;
while (i < 5) {
    System.out.println(i);
    i++;
}
```

6. Class, Interface, methods declaration, and definition. We will explain them in this unit, section 3, Object-Oriented Programming. They shape the structure of our program, creating complex models, that define the modularity and module dependencies in our program. Let us show you an example. Here is an example of a class declaration. Inside you can find some method declarations and definitions:

```
package oppfundamentals;
```

```
public class SUV extends Car{
```

```
    private static final String TYPE_SUV="SUV";
```

```
    public SUV() {
```

```
        super();
```

```
    }
```

```
    public SUV(String color, String brand, String model, double price, double cost) {
        super(color, brand, model, price, cost);
        // TODO Auto-generated constructor stub
    }
```

```
        this.color= color;
```

```
    }
```

```
    @Override
```

```
    public String toString() {
```

```
        return "SUV [color=" + color + ", brand=" + this.getBrand() + ", model="
+ this.getModel() + "];"
    }
```

```
    @Override
```

```
    public String getTypeOfCar() {
```

```
        // TODO Auto-generated method stub
```

```
        return "SUV";
```

```
    }
```

```
}
```

Example of method declaration and definition

```
public String toString() {
```

```
    return "SUV [color=" + color + ", brand=" + this.getBrand() + ", model="
+ this.getModel() + "];"
}
```

As you can see, both classes and methods are made up of block statements.

2.9.7 Introduction to Java Conditions and If Statements

Java supports the usual logical conditions from mathematics:

- Less than: `a < b`
- Less than or equal to: `a <= b`
- Greater than: `a > b`
- Greater than or equal to: `a >= b`
- Equal to `a == b`
- Not Equal to: `a != b`

You can use these conditions to perform different actions for different decisions.

Java has the following conditional statements:

- Use `if` to specify a block of code to be executed, if a specified condition is true
- Use `else` to specify a block of code to be executed, if the same condition is false
- Use `else if` to specify a new condition to test, if the first condition is false
- Use `switch` to specify many alternative blocks of code to be executed

The `if` the conditional statement shows the following structure:

```
if (condition)
    statement;
```

or for a statement block:

```
if (condition) {

    statement1;
    statement2;
}
```

A condition always returns a Boolean value, true or false. We place the condition among parentheses. It can be a simple condition like `age>19`, or a complex condition (using conditional operators), such as `age>19 && age<69`.

We will analyze the Java `if` the sentence in the next example. We will create a new class `IfSentence.java`, and then copy this code:

```
package fundamentals;

import java.util.Scanner;
```

```
public class InputOutputExample {  
  
    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>= 18)  
            System.out.println("The person is an adult");  
  
    }  
}
```

The program is intended to guess whether a person is an adult. As input, it receives the person's age.

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

Assuming that the age is greater than or equal to 18, the program provides as an output that the person is an adult.

```
if (age>= 18) {  
    System.out.println("The person is an adult");  
}
```

However, how does it work? It is simple. First, the logical condition is verified. For instance, if the age receives from the console is age=19, the program validates (19>18) as true. Therefore, the sentence following the if is executed. Run the program, and check for results.

```
Introduce the age of a Person  
19  
The person is an adult
```

If the age introduced is 16, the program will not provide an answer, given that the statement `System.out.println("The person is an adult");` is not executed.

To fully manage both scenarios we can add something to the if sentence. That is the if-else

The if-else conditional statement shows the following structure:

```
if (condition)
    statement1;
else
    statement2;
```

or for a statement block:

```
if (condition) {
    statement1;
    statement2;
} else {
    Statement3;
    Statement4;
}
```

In the case of block sentences, **the trend or convention in Java to write the else is :**

right brace else left brace

} else {

```
if (age >= 18) {
    System.out.println("The person is an adult");
} else {
    System.out.println("The person is a minor age person");
}
```

If (age >= 18) is true the program flow will continue on `System.out.println("The person is an adult");`. Otherwise, `System.out.println("The person is a minor age person");` will be executed.

Nested if else, the else if statement

Let's move and make the example even more complex. We will check this time that the age of the person is greater than 150 it is not a person.

The if-else conditional statement shows the following structure:

```

if (condition) statement1;
else if (condition) statement2;
else statement;

```

,or for statement block:

```

if (condition) {

statement1;
statement2;
}

else if (condition) {

statement3;
statement4;
...
...

}

else {
statement6;
statement7;
...
...

};

```

```

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");

```

Notice, that this time we are integrating three scenarios. There is no limit to keeping adding if-else; Nevertheless, it is considered to be a bad coding practice to have more than five or six if-else together. Thus, developers avoid these long conditional statements decomposing programs.

We can be more specific using conditional operators. Let us explain it. What if we would like to define more complex conditions, for instance, the person can be a teenager. Hence, his age should be between 15 and 18. To fulfill this range, we could write this condition (age >= 15 && age <= 18)

```

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");
```

Activity. Cornell notes:

In this activity, you are required to comment on each line of your code. For instance:

```
// The program asks for the person's age
System.out.println("Introduce the age of a Person");

//it assigns the console input to the age variable
age = sc.nextInt();

//the if sentence verify if the age is greater than or equal to 15
// and less than 18
    if (age>= 15 && age < 18)
```

Practice. Try it yourself

In the example, we need to verify that the person's age is over zero. If not, the program should suggest that the person not be born.

Moreover, you should add retirees to the program people over 65 but less than 150. Try also to add to the program, people in their thirties!

Follow these steps:

1. Write the code
2. Debug the code
3. Comment on the code you have written

2.9.8 Characters and strings

The char type represents individual alphanumeric characters or symbols, like the ones that you type. There are 216 different possible char values, but we usually restrict attention to the ones that represent letters, numbers, symbols, and whitespace characters such as tab and newline. You can specify a char literal by enclosing a character within single quotes; for example, 'a' represents the letter a. For tab, newline, backslash, single quote, and double quote, we use the special escape sequences \t, \n, \\, \', and \", respectively. The characters are encoded as 16-bit integers using an encoding scheme known as Unicode, and there are also escape sequences for specifying special characters not found on your keyboard (see the book site). We usually do not perform any operations directly on characters other than assigning values to variables.

<i>values</i>	characters
<i>typical literals</i>	'a' '\n'

Java's built-in char data type

The String type represents sequences of characters. You can specify a String literal by enclosing a sequence of characters within double quotes, such as "Hello, World". The String data type is not a primitive type, but Java sometimes treats it like one. For example, the concatenation operator (+) takes two String operands and produces a third String that is formed by appending the characters of the second operand to the characters of the first operand.

<i>values</i>	sequences of characters
<i>typical literals</i>	"Hello, World" " * "
<i>operation</i>	concatenate
<i>operator</i>	+

Java's built-in String data type

The concatenation operation (along with the ability to declare String variables and to use them in expressions and assignment statements) is sufficiently powerful to allow us to attack some nontrivial computing tasks. Some typical String expressions in Java can be.

<i>expression</i>	<i>value</i>
"Hi, " + "Bob"	"Hi, Bob"
"1" + " 2 " + "1"	"1 2 1"
"1234" + " " + " " + "99"	"1234 99"
"1234" + "99"	"123499"

Typical String expressions

StringExample.java

Here is our first example of the String type in Java. What is especially useful in Strings is that you can combine them with almost any primitive type using the operator concat "+".

```
package fundamentals;

public class StringExample {

    public static void main(String[] args) {

        String myString= "You can concat almost every primitive type with Strings in
Java.\n";

        double average= 24.1;
        int myAge= 35;
        final boolean STATEMENT_TRUE=true;

        myString = myString + "\n" + "The age average is " + average ;
        myString = myString + "\n" + "Nevertheless, my age is " + myAge ;

        myString = myString + "\n" + "As a result, I am over the average age " + true;

        System.out.println(myString);

    }

}
```

In the example above, we declare four different types of variables: String, double, int, and Boolean.

```
String myString= "You can concatenate almost every primitive type with Strings in
Java.\n";

double average= 24.1;
int myAge= 35;
final boolean STATEMENT_TRUE=true;
```

Java is so flexible with Strings that allows programmers to concatenate these primitive types with Strings. Automatically, as you operate concatenation among Strings and other types, Java offers as a result a new String.

```
myString = myString + "\n" + "The age average is " + average ;
```

The next output results from this sentence's executions. The program will store in MyString

You can concat almost every primitive type with Strings in Java.
The age average is 24.1

Execute the program and check that all types have been transformed to String after using the concat operator. Here is an example of the result:

```
You can concatenate almost every primitive type with Strings in Java.  
The age average is 24.1  
Nevertheless, my age is 35  
As a result, I am over the average age true
```

Ruler.java example

As a more complex example, **Ruler.java** computes a table of values of the ruler function that describes the relative lengths of the marks on a ruler. One noteworthy feature of this computation is that it illustrates how easy it is to craft a short program that produces a huge amount of output.

If you extend this program in the obvious way to print five lines, six lines, seven lines, and so forth, you will see that each time you add two statements to this program, you double the size of the output. Specifically, if the program prints n lines, the n th line contains $2^n - 1$ numbers. For example, if you were to add statements in this way so that the program prints 30 lines, it would print more than 1 billion numbers.

Ruler.java

```
package fundamentals;  
  
public class Ruler {  
    public static void main(String[] args) {  
        String ruler1 = "1";  
        String ruler2 = ruler1 + " 2 " + ruler1;  
        String ruler3 = ruler2 + " 3 " + ruler2;  
        String ruler4 = ruler3 + " 4 " + ruler3;  
        System.out.println(ruler1);  
        System.out.println(ruler2);  
        System.out.println(ruler3);  
        System.out.println(ruler4);  
    }  
}
```



Execution

```

1
1 2 1
1 2 1 3 1 2 1
1 2 1 3 1 2 1 4 1 2 1 3 1 2 1

```

We will learn in this course how to print even the segments above the numbers.

3 Object-Oriented Language

Object-oriented programming is modeled on how, in the real world, objects are often made up of many kinds of smaller objects. This capability of combining objects, however, is only one very general aspect of object-oriented programming. Object-oriented programming provides several other concepts and features to make creating and using objects easier and more flexible, and the most important of these features is that of classes.

A *class* is a template for multiple objects with similar features. Classes embody all the features of a particular set of objects. When you write a program in an object-oriented language, you don't define actual objects. You define classes of objects.

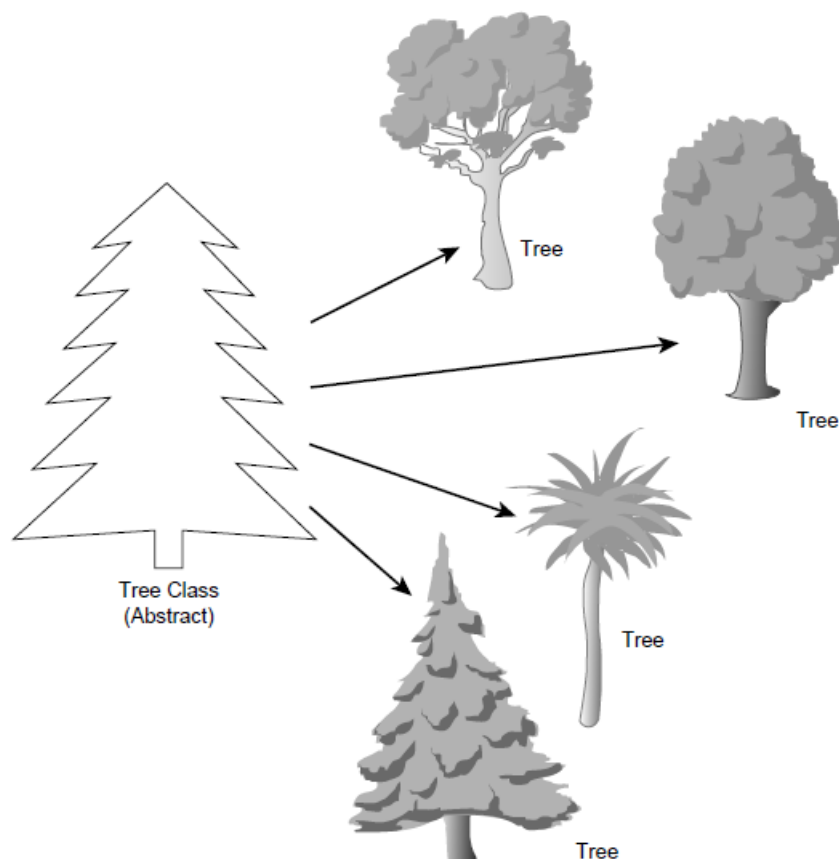
For example, you might have a **Tree** class that describes the features of all trees (has leaves and roots, grows, creates chlorophyll). The **Tree** class serves as an abstract model for the concept of a tree—to reach out and grab, interact with, or cut down a tree you have to have a concrete instance of that tree. Of course, once you have a tree class, you can create lots of different instances of that tree, and each different tree instance can have different features (short, tall, bushy, drops leaves in Autumn), while still behaving like and being immediately recognizable as a tree.

An instance of a class is another word for an actual object. If classes are an abstract representation of an object, an instance is its concrete representation. So what, precisely, is the difference between an instance and an object? Nothing, really. The object is the

more general term, but both instances and objects are the concrete representation of a class.

The terms instance and object are often used interchangeably in OOP language. An instance of a tree and a tree object are both the same thing. In an example closer to the sort of things you might want to do in Java programming, you might create a class for the user interface element called a button.

The Button class defines the features of a button (its label, its size, its appearance) and how it behaves (does it need a single click or a double click to activate it, does it change color when it's clicked, what does it do when it's activated?). Once you define the Button class, you can then easily create instances of that button—that is, button objects—that all take on the basic features of the button as defined by the class, but may have different appearances and behavior based on what you want that particular button to do. By creating a class, you don't have to keep rewriting the code for each individual button you want to use in your program, and you can reuse the **Button** class to create different kinds of buttons as you need them in this program and in other programs.



Pillars of Object-Oriented Programming

Object-oriented programming is based on four pillars, concepts that differentiate it from other programming paradigms. These pillars, some principles we will unveil in later units, rules, and conventions for object-oriented programming originates from the early 90s. Software companies and software engineers pursue high-quality code to deliver better software products and reduce the price of maintenance and manufacturing.

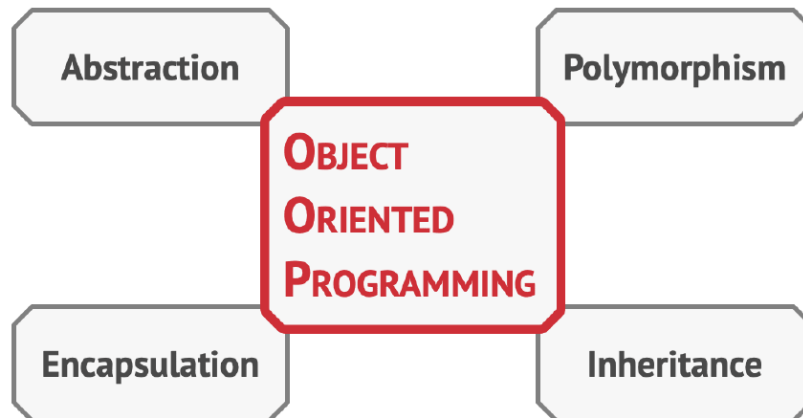
In that search, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides wrote a book ***Design Patterns: Elements of Reusable Object-Oriented Software***. It has been influential to the field of software engineering and is regarded as an important source for object-oriented design theory and practice. More than 500,000 copies have been sold in English and in 13 other languages.

Likewise, many conferences about software engineering were hold in the 90's, to ameliorate software and information system production. We will follow so many of these "rules" that most recognized and admired developers and academics in Computer Science has fixed since that period.

OOP concepts allow us to create specific interactions between Java objects. They make it possible to reuse code without creating security risks or making a Java program less readable.

Here are the four main principles in more detail.

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

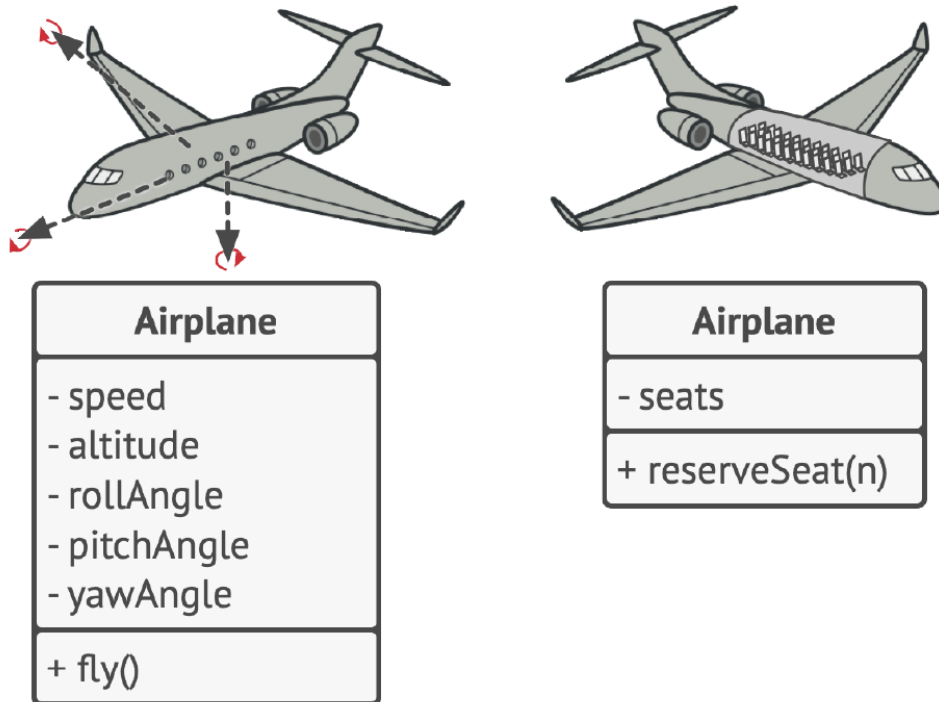


Abstraction

Abstraction is a model of a real-world object or phenomenon, limited to a specific context, which represents all details relevant **to this context with high accuracy and omits all the rest.**

Most of the time when you're creating a program with OOP, you shape objects of the program based on real-world objects. However, objects of the program don't represent the originals with 100% accuracy (and it's rarely required that they do). Instead, your objects only *model* attributes and behaviors of real objects in a specific context, ignoring the rest.

For example, an Airplane class could probably exist in both a flight simulator and a flight booking application. But in the former case, it would hold details related to the actual flight, whereas in the latter class you would care only about the seat map and which seats are available.



Encapsulation

To start a car engine, you only need to turn a key or press a button. You don't need to connect wires under the hood, rotate the crankshaft and cylinders, and initiate the power cycle of the engine. These details are hidden under the hood of the car. You have only a simple interface: a start switch, a steering wheel and some pedals. This illustrates how each object has an interface—a public part of an object, open to interactions with other objects.

Encapsulation is the feature of each module in our code to hide data or operations which are not needed to unveil to other modules of our code. For example, supposing that I have a module call `StoreEmployees` which stores `Employee` Data. Other modules of our application, such as windows (the graphical interface) will use this module to save data. Nevertheless, this graphical interface does no need to know whether `StoreEmployees` save the data in a `File` or a `DataBase`. Those implementation details are hidden, encapsulated. Thus, the `Window` needs only a function `storeEmployeeData(Employee data)` to use this functionality.

Encapsulation in Object-Oriented programming is the ability of an object to hide parts of its state and behaviors from other objects, exposing only a limited interface to the rest of the program.

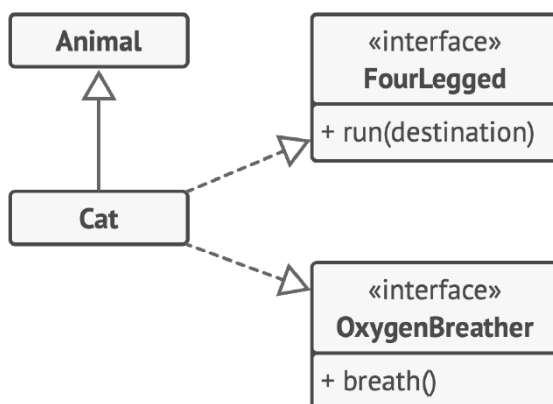
To *encapsulate* something means to make it `private`, and thus accessible only from within

There is a little bit less restrictive mode called **protected** that makes a member of a class available to subclasses as well. Interfaces and abstract classes/methods of most programming languages are based on the concepts of abstraction and encapsulation.

Inheritance

Inheritance is the ability to build new classes on top of existing ones. The main benefit of inheritance is code reuse. If you want to create a class that's slightly different from an existing one, there's no need to duplicate code. Instead, you extend the existing class and put the extra functionality into a resulting subclass, which inherits fields and methods of the superclass.

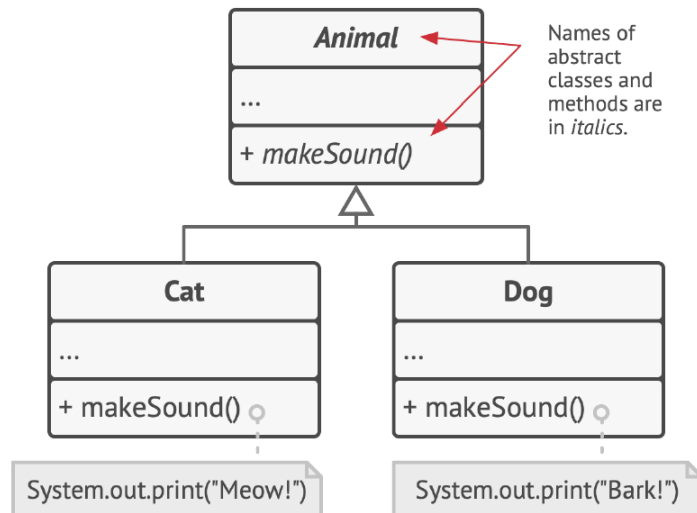
The consequence of using inheritance is that subclasses have the same interface as their parent class. You can't hide a method in a subclass if it was declared in the superclass. You must also implement all abstract methods, even if they don't make sense for your subclass.



Polymorphism

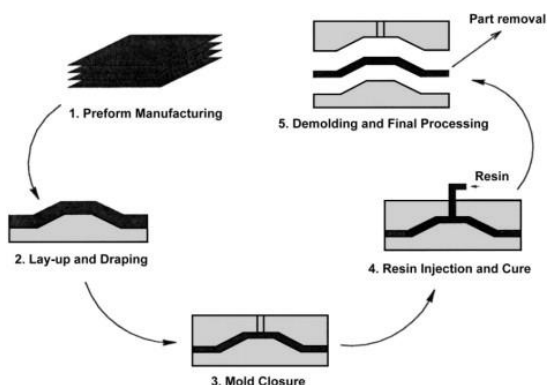
The word polymorphism comes from the Greek words for "many shapes". A polymorphic method, for example, is a method that can have different shapes, where "shape" can be regarded as type or behavior.

Let's look at some animal examples. Most **Animals** can make sounds. We can anticipate that all subclasses will need to override the base **makeSound** method so each subclass can emit the correct sound; therefore we can declare it *abstract* right away. This lets us omit any default implementation of the method in the superclass, but force all subclasses to come up with their own.



3.1 Java as an Object-Oriented programming

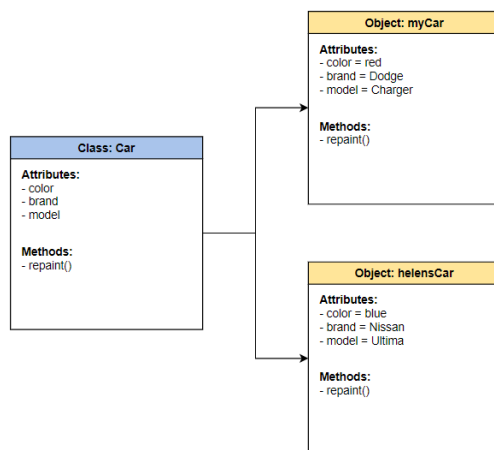
Java is Object Oriented in nature. It means that programs are made up of objects. Nevertheless, to build or create objects, you would need a blueprint, a template. An analogy to describe classes and objects could be taken from manufacture. When you need to build one hundred thousand door handles for cars, you should follow some steps. First, you should design the car body-work. Then you build a molde for that piece. And finally, you should go for mass production.



To program in a Object Oriented language, you need your mold, your template to create a numerous set of objects. For example, say we created a class, `Car`, to contain all the properties a car must have, `color`, `brand`, and `model`. You then create an instance of a `Car` type object, `myCar` to represent my specific car.

We could then set the value of the properties defined in the class to describe my car, without affecting other objects or the class template. We can then reuse this class to represent any number of cars.

We already describe this action. It is the first pillar in OOP. **Abstraction is a model of a real-world object or phenomenon**. We have abstracted the real world car, in a coding representation call `Car` class. Given that we are representig the car via software, we add the properties and the behaviors we need to control in our program. Supposing that we are selling new cars, we do not need to store the odometer, which pinpoint the number of kilometers of our car. We have made the decision to leave out of our representation this feature.



Let us create this project `ObjectOrientedProgrammingFundamentals` in Eclipse. We will include this two classes, but first add the `MainClass.java`.



```
package oppfundamentals;
```

```
public class MainClass {
```

```
    public static void main(String[] args) {
        // TODO Auto-generated method stub
    }
}
```

```
        System.out.println("Hello there. Howdy. My first java program on the  
running");  
    }  
}
```

Every source code file in java contains at least a class with the public modifier. Therefore, to program in Java we need to add classes to our code. To explain object and classes the first thing, we will do is adding one to this project. Please, add the class car to the project, and copy the following code to the file.

```
package oppfundamentals;  
  
public class Car {  
    private String color;  
    private String brand;  
    private String model;  
  
    public Car (String color, String brand, String model) {  
        this.color=color;  
        this.brand=brand;  
        this.model = model;  
    }  
  
    public void rePaint (String color) {  
        this.color=color;  
    }  
  
    @Override  
    public String toString() {  
        return "Car [color=" + color + ", brand=" + brand + ", model=" + model +  
        "];"  
    }  
}
```

Let us break down the class in each different element. Highlighted in yellow, you can find **class properties or attributes**. They are intended to store information about each individual car. This information is what we call the state of the object. For instance, Assuming that the car is red, its present state is being red. If we repaint the car we change its state. Class properties behave like variables we studied in Unit 1 and in this document. They have a **Type** and a **name**. In this example the **type** is String.

```
private String color;  
private String brand;  
private String model;
```

In the former example, the class has three properties: color, brand and model. Properties as you can see, describe the car. This is less than a half of an object. The other half is what the object can do, its behavior. The car in our program can be repainted. To provide objects with behavior, that is the actions they can take, in the classes we can define methods. You could infer that methods are similar to subroutines. You are right, they are almost the same. However, these functions belong to objects, and you can only apply them through objects.

The first function you usually find in a class is the constructor. This is very the most unique method because it allows programmers to create objects as we will explain later. It is very particular since it does not have a return type. The return type is implicit, an object of this class.

```
public Car (String color, String brand, String model) {  
    this.color=color;  
    this.brand=brand;  
    this.model = model;  
}
```

```
public NO RETURN TYPE HERE Car (String color, String brand, String model) {
```

After the constructor, programmers usually write methods to add an operation or a behavior to the object. To add operations to a class we use methods. Remember that methods can be procedures or functions. The **repaint method** let the car be repainted. It changes the car behavior. This is a feature intrinsic to the car. A person cannot be repainted, nor the sea.

```
public void rePaint (String color) {  
  
    this.color=color;
```

}

Finally, the `toString()` method returns a description of the object contents in a `String`. Java use the `toString` method to be able to print the method in the output. Otherwise, it would print a memory address, as we explain later.

```
@Override
    public String toString() {
        return "Car [color=" + color + ", brand=" + brand + ", model=" + model +
    "];";
    }
```

Overall, there are some methods that allow create objects, others let us do actions, calculations or return processed data. Nevertheless, we are speaking about objects, but we do not have one. We have a blueprint, a mold, but we have not created and object yet.

3.1.1 The `this` operator, reference to Object vs reference to class

When your coding inside a class, we have at our disposal a tool, or a reserved key to accede to the object anywhere. It is called the `this` operator. This works similar to a variable declared as `car`, `Car myCar` which provides access to all properties and methods of the object. Actually it points out to the object you have created for this class.

```
public abstract class Car {
    protected String color;
    private String brand;
    private String model;

    public Car (String color, String brand, String model, double price, double cost)
    {

        this.color=color;
        this.brand=brand;
        this.model = model;
    }
}
```

`This.color` refers to the `protected String color;` property. In doing so you can distinguish between the parameter `String color` and the property `protected String color;` You should take into consideration that the `this` operator can be used only inside the class code. In addition, the `this` operator refers to an object you have create, not to the class. **It is nor working unless an object has been created. We cannot use it in Static**

To make reference to a class you should write the name of the class, usually followed by a static constant or static method. Change the last line of your Car constructor, adding the Class reference to the static method. It makes the code more legible and cleaner.

```
Car.incrementNumberOfCarsCreated();

public Car (String color, String brand, String model{

    this.color=color;
    this.brand=brand;
    this.model = model;

    Car.incrementNumberOfCarsCreated();

}
```

3.2 Creating objects

Let us back to the main program. We will create a car object. The process is called instantiation or instantiating an object; Hence, an object is an instance of a class.

First, we need to define a program variable of this class. Second, we need to call the class constructor using a reserved key word, **new**. Creating an object is easy, thus add the next two lines after the System.out.println sentence

```
public static void main(String[] args) {
    // TODO Auto-generated method stub

    System.out.println("Hello there. Howdy. My first java program on the
running");

    //Add the lines after this

    Car myTesla = new Car("red", "Tesla", "Model S High Performace");

    System.out.println(myTesla);
```

Any time you type **new** followed by a call to the class constructor, we create an object of this car. Moreover, you can create as many objects as you need. Let's do it, include the next car to your program.

```
Car myBMW = new Car("blue", "BMW", "i5");  
  
System.out.println(myBMW);
```

To use some of the car behavior in your program, repaint the BMW in white.

```
myBMW.repaint("white");  
  
System.out.println("MYBMW after repaint " + myBMW);
```

Run your program and you might get a similar outcome as :

```
Hello there. Howdy. My first java program on the running  
Car [color=red, brand=Tesla, model=Model S High Performance]  
Car [color=blue, brand= BMW, model=i5]  
MYBMW after repaint Car [color=white, brand= BMW, model=i5]
```

Run the program again, but first erase the toString method from the car class, we will added again after this executions.

The result of taking out the toString Method is this execution:

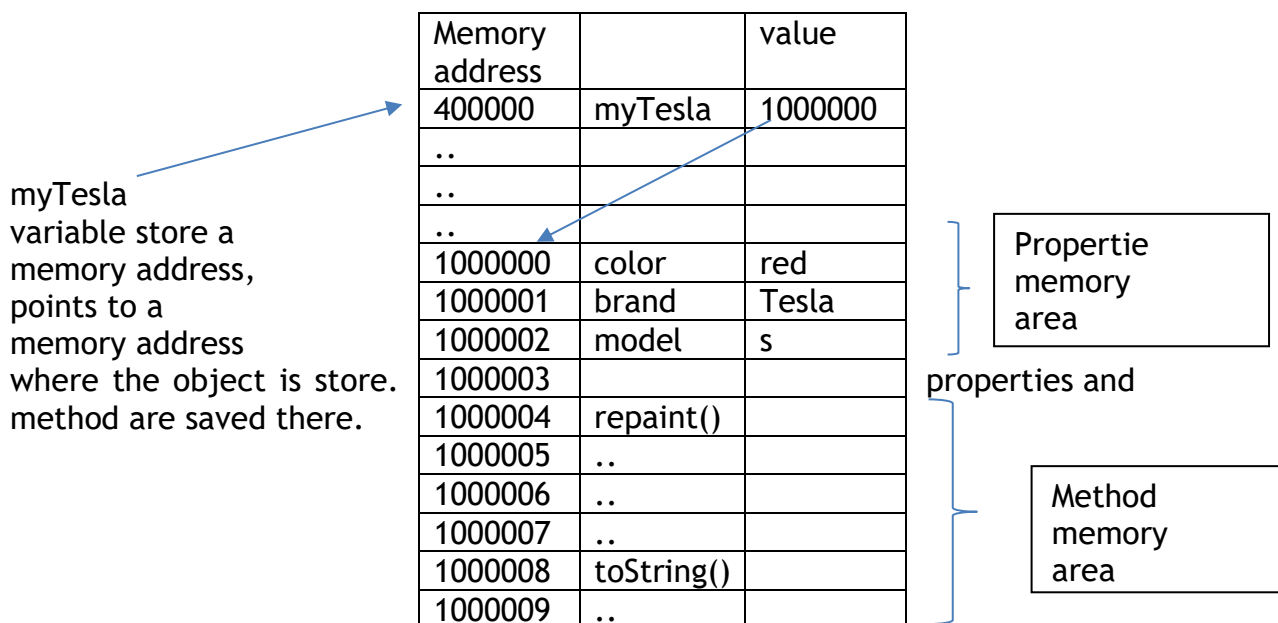
```
oppfundamentals.Car@3830f1c0  
oppfundamentals.Car@39ed3c8d  
MYBMW after repaint oppfundamentals.Car@39ed3c8d  
color Property access via Method White
```

Now the println, print the real content of a object variable, which is a memory address in hexadecimal. Undo the toString delete, and leave the class in its initial state. This exercise serve as an introduction to the next question

But, What happen when you create an object in java?

An object variable is a pointer to a memory location, in this memory location it is saved all the object information, properties and code.

RAM Memory



MyTesla variable will store itself a memory position. The memory where the object starts, in this case 1000000.

3.2.1 Public vs private properties. Encapsulation

For object properties Java offers public methods to be used by other objects, yet it can offer properties. Let us change the property modifiers of car we declared before.


```
public class Car {  
    public String color;  
    public String brand;  
    public String model;  
    public static final String TAG = "Car Object";  
}
```

and now add this line at the end of your main program.

```
System.out.println("color Property access " + myBMWi.color);
```

Since we have changed the property declaration, and we must add the public modifier, now you can accede to the color attribute using the dot “.” Operator. This operator provides access to any public declaration you have added to your classes. It means that through the object variable(after being created) you can accede to its state, properties, or its methods. The variable object is a variable that at the same time contains a property, kind of a variable.

Since the lately 90’s, it is an object-oriented programming convention and a proper use of languages like java not to declare **properties as public following Encapsulation**. Encapsulation lays behind some measurements or evidence from developer’s experience:

- Your code can be more prone to errors: Statically, another object altering other object’s state will lead your program to the risk of multiple errors so that it is less predictable.
- It put at risk your object data integrity, resulting in your posing a threat to your application data integrity.
- It makes the code less legible and maintainable.

To conclude, **it is almost forbidden to declare properties public to produce clean code.**

We apply the second Pillar of Object-Oriented Programming

Although sometimes your object should offer constant variables to other, this practice is allowed and recommendable. For example, the constant variable TAG is properly declared in the event other objects need it.

```
public static final String TAG = "Car Object";
```

Clean code is code that is easy to read, understand and maintain. Clean code makes

software development predictable and increases the quality of a resulting product.

Henceforth, we will not declare properties public.

The idea is that object data can be accessed via methods. Data can be encapsulated such that it is invisible to the "outside world".

For more details on how to obtain quality software and refactor your code, you can review this link:

<https://refactoring.guru/refactoring>

That gives rise to two ways of approach classes:

- A class is complex type in Java. We have declared a variable `Car myBMW1`. And this type is made up of other variables, its properties. From a programmer point of view, a class is a Type
- Moreover, a class is an abstraction of a real object. This properties are what define and represent the car features. this object in our program. For a software designer, a class is reality realization, to be managed by our software.

Methods. Encapsulation II. Private vs Public

In the previous section it has been covered, object data, its state. Using methods, we provide objects with operations, a behavior. We already study that a method can be a procedure which performs an action but do not give back a result, or a function that return results.

We can distinguish two blocks or parts in a method:

The signature or header of a function. It is also know as the interface of the method. The arity is the list of parameters.

Signature or header



```
public void repaint (String color)
```

↑
modifier

function body

statements

```
{
    this.color=color;
}
```

The Method repaint doesn't return values, thus it is a procedure

```
public void repaint (String color) {
    this.color=color;
}
```

Returning type

method name

parameters

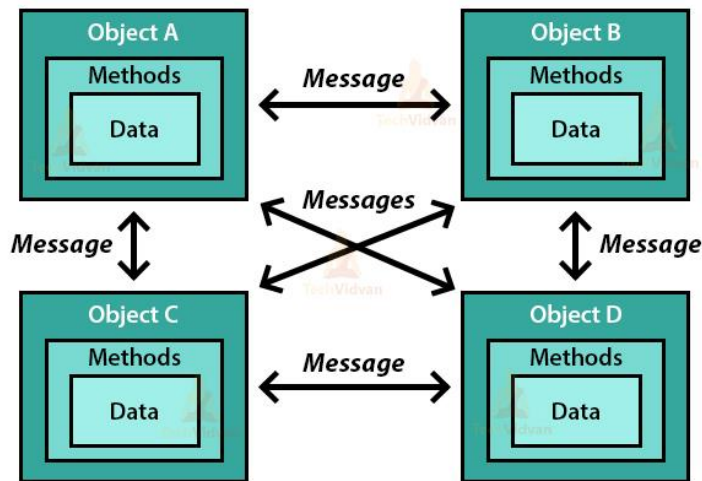
```
public String toString()
```

return statement

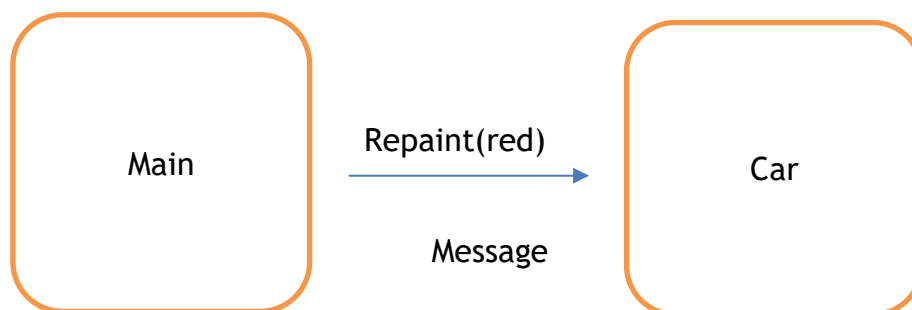
```
{
    return "Car [color=" + color + ", brand=" + brand + ", model=" + model +
    "];"
}
```

In this case, the method toString is a catalogue as a function. Conversely, it does not receive parameters. You could have also methods function alike that receive parameters. But there is another approach for methods. They can be considered as tools to receive messages from other objects and send messages to other objects.

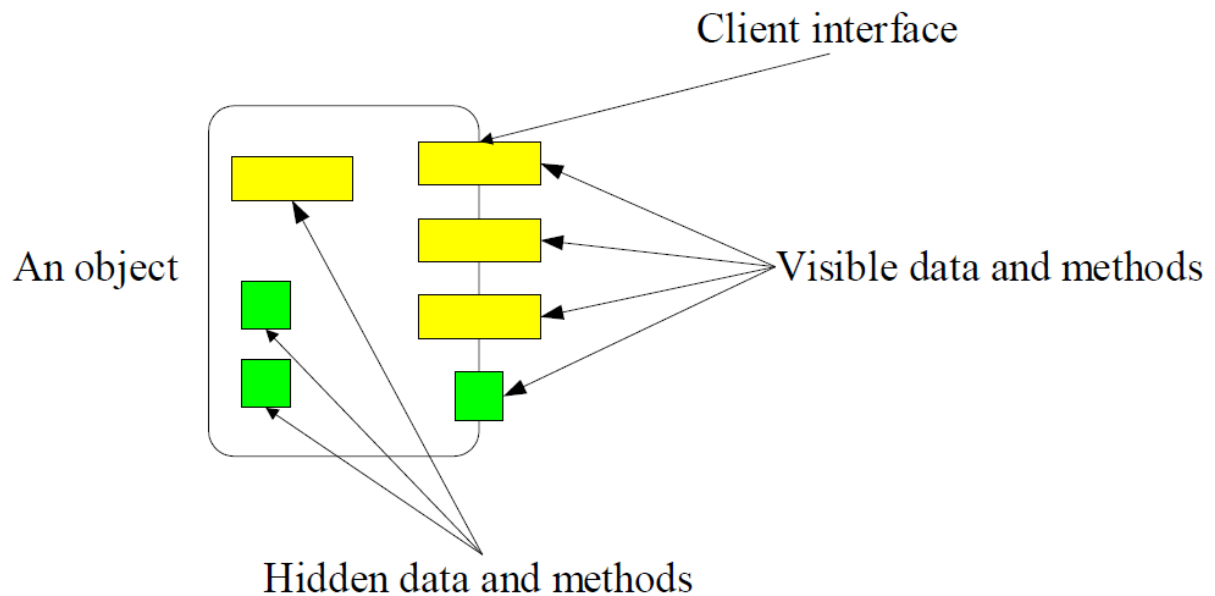
Message Passing



My Main class send a message to the car, to get repainted in red.

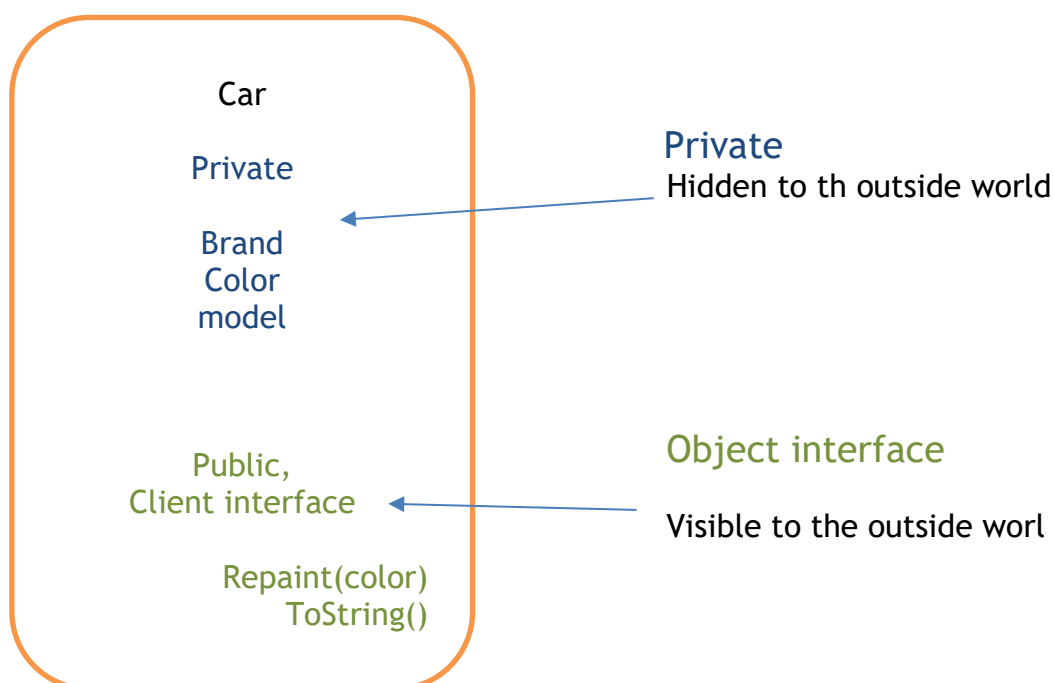


Likewise, methods can be a mean to give access to objects public data. What we are trying to achieve through encapsulation is hide the private object data and behavior which is not needed for other objects. Moreover, with encapsulation we are trying to offer a clear interface of the class. As a result, our objects will offer the necessary methods, “messages”, to provide its full functionality, and hide which is not convenient to show, the object internal working.



What the "outside world" cannot see it cannot depend on!

- The object is a "fire-wall" between the object and the "outside world".
- The hidden data and methods can be changed without affecting the "outside world".



Accessor Methods

It is clear that with the present design of our class, we do not have access to valuable data, such as brand, model or color. Although we must declare our properties as private, programmers are committed to offer means to provide access to the object data. To do so we must implement that what we call **accessor methods**. These **methods provide access and allow make changes in the object data** that should be public.

Let us add to the Car class these methods. They are known as getters and setters owing that they allow other objects in our program to modify our program properties.

```
public String getColor() {  
    return color;  
}  
  
public void setColor(String color) {  
    this.color = color;  
}  
  
public String getBrand() {  
    return brand;  
}  
  
public void setBrand(String brand) {  
    this.brand = brand;  
}  
  
public String getModel() {  
    return model;  
}  
  
public void setModel(String model) {  
    this.model = model;  
}
```

We have extended the new object interface

Private

Hidden to the outside world



Object Interface

Visible to the outside world



From this point on, we are to extend our class functionality, adding new features to it.

Add two new properties and change the class constructor:

```
public class Car {  
    private String color;  
    private String brand;  
    private String model;  
    private double price;  
    private double cost;  
  
    public static final String TAG = "Car Object";  
  
    public Car (String color, String brand, String model, double price, double cost) {  
  
        this.color=color;  
        this.brand=brand;  
        this.model = model;  
        this.price=price;  
        this.cost=cost;  
  
    }  
}
```

Given that we do not want to allow other objects to modify or accede to these new properties, do not add new getters and setters for these properties. Instead, we will add the next methods before the toString method. We are only interested in offering to other object the profit made by each car sale.

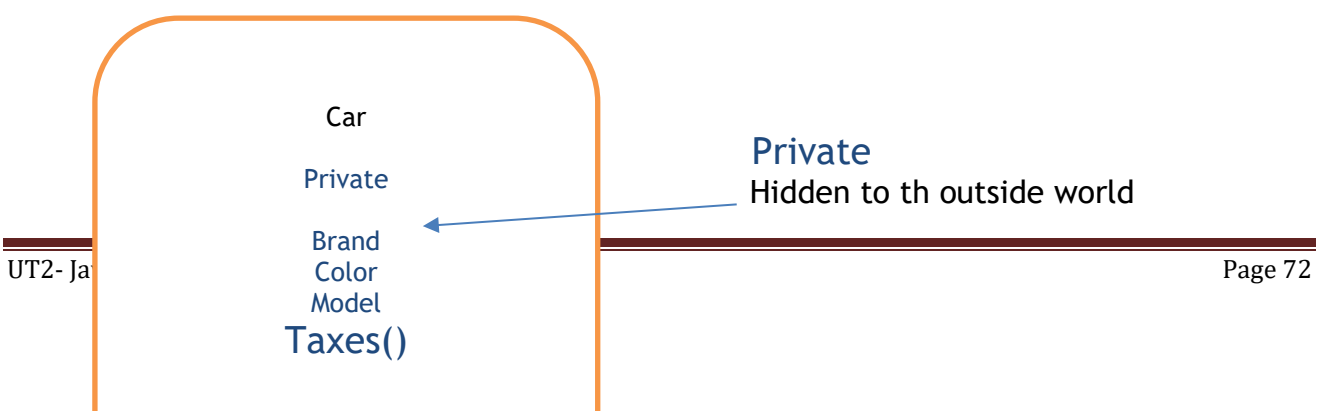
```
public double profits(double vanRate) {  
    return (price-cost) -taxes(vanRate);  
}  
  
private double taxes( double vanRate) {  
    return (price-cost)* vanRate;  
}
```

Otherwise, we do not want to explain how we calculate taxes. On the one side, the taxes method is intended to internal calculations. Consequently, we declare it private, provided that it will be inside the class. To the external world we are only offering the profits method.

On the other side, the profits method is aimed to offer a calculation to other objects, profit per sale. The public methods we have defined before provide access to the object state. Conversely, the profits method offers a simple profit algorithm, a behavior.

These methods, whose aim is providing internal calculations and algorithms, are labeled as **Private Methods** in contrast to **Accesor Methods**, which give us access to the class state or properties. Also, we can add to the equation, **Behaviour Methods**, which offer a functionality to the external world.

The new object definition



Interface

Visible to the outside world



To verify our program updates add this code at the end of your main program to test new methods and check for results.

```
public static void main(String[] args) {  
    // TODO Auto-generated method stub  
  
    System.out.println("Hello there. Howdy. My first java program on the  
running");  
  
    Car myTesla = new Car("red", "Tesla", "Model S High Performace",  
50000, 30000);  
  
    System.out.println(myTesla);  
  
    Car myBMWi = new Car("blue", " BMW", "i5", 50000, 40000);  
  
    System.out.println(myBMWi);  
  
    myBMWi.rePaint("white");  
  
    System.out.println("MYBMW after repaint " + myBMWi);  
  
    System.out.println("color Property access via Method " +  
myBMWi.getColor());  
  
    System.out.println("Income from BMW sale:" + myBMWi.profits(0.10));  
}
```

```
System.out.println("Income from Tesla sale:" + myTesla.profits(0.10));
```

Here the execution:

```
Hello there. Howdy. My first java program on the running
Car [color=red, brand=Tesla, model=Model S High Performace]
Car [color=blue, brand= BMW, model=i5]
MYBMW after repaint Car [color=white, brand= BMW, model=i5]
color Property access via Method white
Income from BMW sale:9000.0
Income from Tesla sale:18000.0
```

Although we declare variables and methods in our class, methods and properties belong to objects, are part of each object. It is worth noting that each object has its own value for the color property , Tesla color=red, and BMW color=blue. Subsequently, each object has its own methods. The BMW object method `myBMWi.profits(0.10)` return value is 9000, whereas the Tesla object method `myTesla.profits(0.10));` gives as a result 18000. Even If the algorithm is the same, the profit method depends on objects properties. The class is only a template, objects copy the code (properties and methods) as long as they are created. It may be infer that any time we use no modifier or the public, private or protected modifier, we create a blueprint of a property or method to be use for objects.

To conclude, in a class you can distinguish among these types of methods:

- Constructor method
- Private Methods
- Accesor Methods
- Behavior Methods or operations

3.2.2 Static properties and methods.

In the two previous sections we have deal with properties and methods. In our classes, we defined properties and methods to be part of an object, an class instances. That translates in the ability of objects to store their data or keep their state. Moreover, these mechanisms, methods, offer an interface, an entry point, to receive messages from other objects, or to perform an operation.

Nevertheless, Java, as in other OPP languages, provides the mechanism to create class properties and methods. The static modifier allows programmers to define properties and methods that belongs to the class. Object instances do not replicate them, but they have access to them, and multiple objects of the same class share these static methods and properties. To prove so, we are to add some code to our class

We have made some additions to the Car Class, highlighted in yellow. Let us list and illustrate changes:

- We have introduced a static property, a class property `numberOfCars`.

```
private static int numberOfCars = 0;
```

- We have included a static method that returns the number of cars created, which make use of the static property `numberOfCars`. The other increments the number of cars created by one **In static methods we can only use static properties**

```
public static int numberOfCarsCreated() {  
    return numberOfCars;  
}
```

```
public static void incrementNumberOfCarsCreated() {  
    numberOfCars = numberOfCars + 1;  
}
```

- We have modified the constructor to add one unit to `numberOfCars` anytime a car is created. That update results in keeping an accountability of cars created in our program

```
public Car (String color, String brand, String model, double price, double cost) {  
    this.color=color;  
    this.brand=brand;  
    this.model = model;  
  
    this.price=price;  
    this.cost=cost;  
  
    incrementNumberOfCarsCreated();  
}
```

```
public class Car {  
    private String color;  
    private String brand;  
    private String model;  
    private double price;  
    private double cost;  
  
    public static final String TAG = "Car Object";  
    private static int numberOfCars =0;  
  
    public Car (String color, String brand, String model, double price, double cost) {  
  
        this.color=color;  
        this.brand=brand;  
        this.model = model;  
        this.price=price;  
        this.cost=cost;  
  
        incrementNumberOfCarsCreated();  
  
    }  
  
    public String getColor() {  
        return color;  
    }  
  
    public void setColor(String color) {  
        this.color = color;  
    }  
  
    public String getBrand() {  
        return brand;  
    }  
  
    public void setBrand(String brand) {  
        this.brand = brand;  
    }  
  
    public String getModel() {  
        return model;  
    }  
  
    public void setModel(String model) {  
        this.model = model;  
    }  
  
    public void rePaint (String color) {
```

```
        this.color=color;
    }

    public double profits(double vanRate) {
        return (price-cost) -taxes(vanRate);
    }

    private double taxes( double vanRate) {
        return (price-cost)*vanRate;
    }

    public static int numberOfCarsCreated() {
        return numberOfCars;
    }

    public static void incrementNumberOfCarsCreated() {
        numberOfCars = numberOfCars + 1;
    }
}
```

In memory, these static properties and methods are store in an Area reserved for the Car Class. Nevertheless, since myTesla and myBMW contain objects that are member of the Car Class, they have permission to accede and use these static components.

Moreover, given than `numberOfCarsCreated()` has been declared public any object or class in your program can accede to them. On the contrary, `incrementNumberOfCarsCreated()`, declared as private, can be use by members of the class.

RAM Memory

Memory address		value
200000	Class car	

200001	<code>numberOfCars</code>	0
	<code>numberOfCarsCreated()</code>	
	<code>incrementNumberOfCarsCreated()</code>	
40000	myTesla	1000000
40001	myBMW	1200004
..		
..		
..	MyTesla object	
1000000	color	red
1000001	brand	Tesla
1000002	model	s
1000003		
1000004	repaint()	
1000005	..	
1000006	..	
1000007	..	
1000008	toString()	
1000009	..	
...		
...	MyBMW object	
1200004	color	blue
1200005	brand	BMW
1200006	model	I5
	

You can now try these changes in your program and verify that they work as they were aimed. Add this line to the end of the main function.

```
System.out.println("Total cars created: " + Car.numberOfCarsCreated());
```

After running the program and having creating two cars, `numberOfCarsCreated()` answer correctly.

```
Car [color=red, brand=Tesla, model=Model S High Performace]
Car [color=blue, brand= BMW, model=i5]
MYBMW after repaint Car [color=white, brand= BMW, model=i5]
color Property access via Method white
Income from BMW sale:9000.0
Income from Tesla sale:18000.0
Total cars created: 2
```

If you examine the statement `Car.numberOfCarsCreated()`, to call the static method `numberOfCarsCreated()`, you need to first type the name of the class, `Car`.

Try to call from main `Car.incrementNumberOfCarsCreated()`. The compiler will prompt an error. Could you infer why? Deliver a conjecture.

Finally, let us review your code. We were talking about constant variables in a class in previous sections. Why do we declare this constant variable with the static modifier? Do you think you can use it in your main program? Supossing that you can, do it.

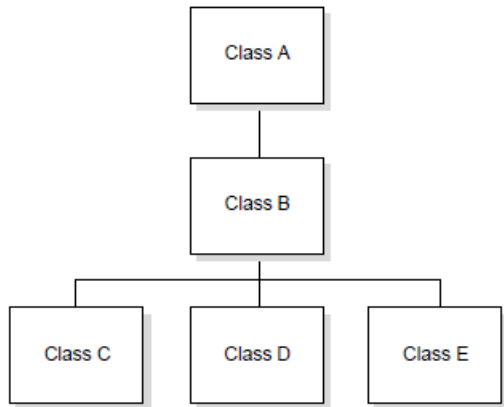
```
public static final String TAG = "Car Object";
```

3.3 Inheritance in Java

Inheritance is one of the most crucial concepts in object-oriented programming, and it has a very direct effect on how you design and write your Java classes. Inheritance is a powerful mechanism that means when you write a class you only have to specify how that class is different from some other class, while also giving you dynamic access to the information contained in those other classes.

With inheritance, all classes—those you write, those from other class libraries that you use, and those from the standard utility classes as well—are arranged in a strict hierarchy.

Each class has a superclass (the class above it in the hierarchy), and each class can have one or more subclasses (classes below that class in the hierarchy). Classes further down in the hierarchy are presumed to inherit from classes further up in the hierarchy.



A class hierarchy. • Class A is the superclass of B

- Class B is a subclass of A
- Class B is the superclass of C, D, and E
- Classes C, D, and E are subclasses of B

Subclasses inherit all the methods and variables from their superclasses—that is, in any particular class, if the superclass defines behavior that your class needs, you don't have to redefine it or copy that code from some other class. Your class automatically gets that behavior from its superclass, that superclass gets behavior from its superclass, and so on all the way up the hierarchy.

Your class becomes a combination of all the features of the classes above it in the hierarchy. At the top of the Java class hierarchy is the class `Object`; all classes inherit from this one superclass.

`Object` is the most general class in the hierarchy; it defines behavior specific to all objects in the Java class hierarchy. Each class farther down in the hierarchy adds more information and becomes more tailored to a specific purpose. In this way, you can think of a class hierarchy as defining very abstract concepts at the top of the hierarchy and those ideas becoming more concrete the farther down the chain of superclasses you go.

Most of the time when you write new Java classes, you'll want to create a class that has all the information some other class has, plus some extra information. For example, you may want a version of a `Button` with its own built-in label. To get all the `Button` information, all you have to do is define your class to inherit from `Button`.

Your class will automatically get all the behavior defined in `Button` (and in `Button`'s superclasses), so all you have to worry about are the things that make your class different

from Button itself. This mechanism for defining new classes as the differences between them and their superclasses is called subclassing.

Now you will try to create your own hierarchy creating new classes that inherit from Car. Indulge yourself creating new subclasses from Car.

The first we will add to the pack is SUV.

```
public class SUV extends Car{

    public SUV(String color, String brand, String model, double price, double cost) {
        super(color, brand, model, price, cost);
        // TODO Auto-generated constructor stub
    }

}
```

To achieve inheritance, you should use the extends reserved word followed by the name of the class, `extends Car`.

If you go over the Constructor, you will realize there is something new, a call to super. Super is a reserved word aimed for classes. It purposes is calling the immediate father constructor. As a result, the parent or super class constructor code will be executed. Then This code will be executed:

```
this.color=color;
    this.brand=brand;
    this.model = model;
    this.price=price;
    this.cost=cost;

    incrementNumberOfCarsCreated();
```

One of the many advantages of inheritance is code reusability. We do not need to write again all the superclass code. As SUV inherits from Car, all car Code is at his disposal.

Let's continue creating a new SUV car. But first, we will add a new MainClass, call AppInheritance.java.

```
package oppfundamentals;
```

```
public class AppInheritance {
```

```

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        SUV myTeslaSUV = new SUV("red", "Tesla", "Model S High Performace",
50000,30000);

        Car.numberOfCarsCreated();

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

        System.out.println("Total cars created: " + Car.numberOfCarsCreated());

        myTeslaSUV.rePaint("White");

        System.out.println("My new SUV repainted:" + myTeslaSUV.toString());

    }
}

```

The running result should be something like:

```

My new SUV:Car [color=red, brand=Tesla, model=Model S High Performace]
Total cars created: 1
My new SUV repainted:Car [color=White, brand=Tesla, model=Model S High Performace]

```

“Total cars created:1” results from myTeslaSUV being a SUV, which is a car since it inherits from th Car Class. In the SUV constructor, we call to super, the superclass constructor, which count the numbers of car created.

Heed your attention now to this method call : `myTeslaSUV.toString()`. Even if this method is not codified in the SUV class it can use it, because it inherits all superclass code.

Protected vs private

It will be introduced here the protected modifier. The best choice to describe the protected modifiers is through use. Hence, you should introduce this code to the SUV constructor

```

public SUV(String color, String brand, String model, double price, double cost) {
    super(color, brand, model, price, cost);
}

```

```
// TODO Auto-generated constructor stub
```

```
    this.color= color;  
}
```

The statement highlighted would cause a compiler error that results from the property color declaration. Providing that you have declared the **color property as private, inheriting subclasses cannot accede to it**. It seems to be awkward that SUV has inherits the Car class code but it can accede straight to properties. Modifiers in Java provide levels of security about code access.

Nevertheless, sometimes you will need to manipulate superclass properties in your subclass. It is not the most convenient procedure in OOP although you can change the color property modifier to protected to achieve so.

```
public class Car {  
    protected String color;  
    private String brand;  
    private String model;  
    private double price;  
    private double cost;  
}
```

Check up that your code is compiling

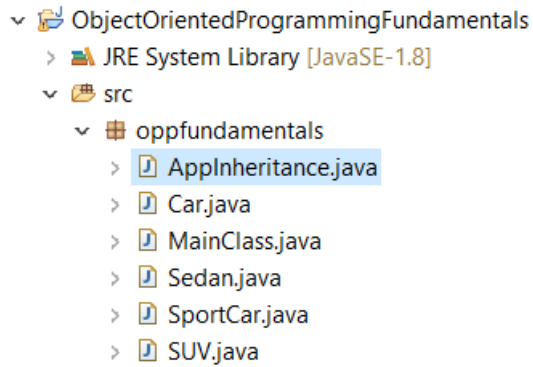
¿Why protected and not public?

Programmer use protected instead of public, when they need to use a property from a superclass, but they do not want to make it public to all classes in their class model. Access from SUV to the property color is granted because of the protected modifiers. Even a subclass of SUV, such as Electric SUV could have access to the color property since the protected modifier grant access to all subclasses in the hierarchy, direct and indirect inheritance include. **It is not a good practice, but sometimes is unavoidable.**

Extending the inheritance hierarchy

It is time to add some more classes to your hierarchy. Therefore, create the class CrossOver, Sedan and SportCar which inherit from Car.

Your project class model should look like the next figure:



Remember to make them extends from Car. Finally add the SUVElectric.java to the hierarchy. Unlike the others, SUVElectric will not inherit from Car, instead it will extends from SUV

```
package oppfundamentals;
```

```
public class SUVElectric extends SUV{
```

```
    public SUVElectric(String color, String brand, String model, double price, double  
cost) {
```

```
        super(color, brand, model, price, cost);
```

```
    }
```

```
}
```

Our class hierarchy

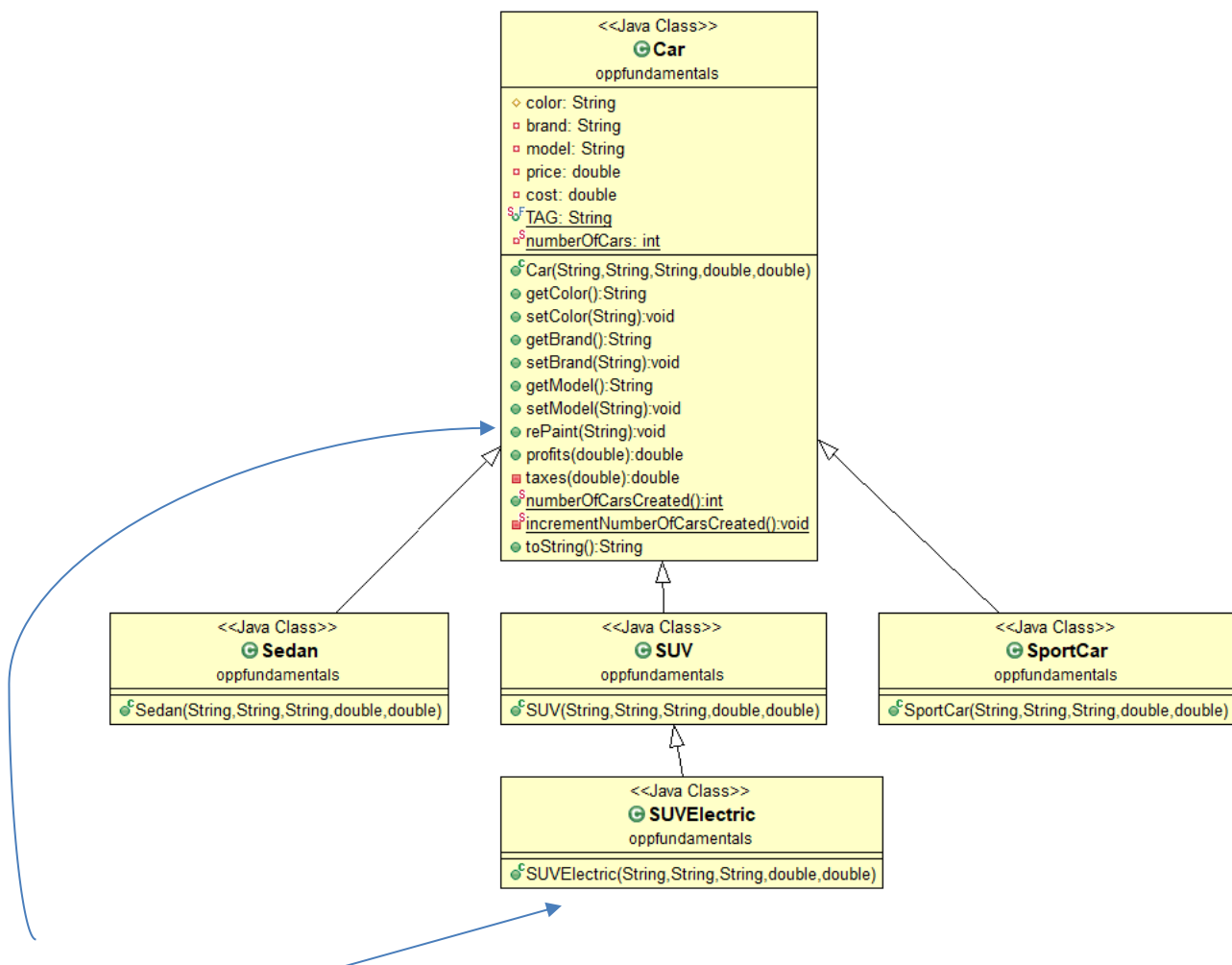


Notice that SUVElectric does not inherit directly from Car. Nevertheless, through SUV Car it will inherit the Car Class code. Moreover, it will inherit any addition to SUV that it is not included in Car. Try to manipulate the color property in the SUVElectric class so that you can verify that it is plausible.

Pay attention to methods usage. When we call the method profits in SUVElectric

In the event that SUV Electric call the method repaint, the call is redirected to the method definition in the class Car.

```
Car myTeslaSUVElectric = new SUV("red", "Tesla", "Model S High Performace", 50000, 30000);
myTeslaSUVElectric.rePaint("green");
```



```
myTeslaSUVElectric.rePaint("green");
```

3.4 Polymorphism

One of the main achievements or most advanced characteristics of OPP is Polymorphism. It means that similar objects respond similarly to the same message, which can be translated to objects in a hierarchy offer the same method, for example the toString method we introduced before. Nevertheless, this method implementation could slightly differ from class to class.

For example, consider a drawing program that lets the user draw lines, rectangles, ovals, polygons, and curves on the screen. In the program, each visible object on the screen could be represented by a software object in the program. There would be five classes of objects in the program, one for each type of visible object that can be drawn. All the lines would belong to one class, all the rectangles to another class, and so on. These classes are obviously related; all of them represent “drawable objects.” In contrast, the way they are drawn is different. The draw() method should differ from subclass to subclass, even though all of them are drawable objects.

Two concepts must be introduced here

Overloading

One of the techniques to achieve something similar to polymorphism in our classes is applying overloading. The core idea of overloading is providing multiple implementations of the same method. What lets the Java compiler differentiate versions is the method arity, the type and number of parameters each version of the method defines. Obviously, we cannot write two versions of the same method with the same arity in the same class.

Attribute names of a class may be the same as those in another class since they are accessed independently. An attribute *x* in a class does not necessarily have any semantic bearing with another as they have different scopes, and does not preclude using the same attribute there.

Within a Java class construct, methods may share the same name as long as they may be distinguished either by:

- the number of parameters, or
- different parameter types.

This criterion requires a message with associated parameters to be uniquely matched with the intended method definition.

We will include this new version of the method `taxes(double vanRate, double localTaxes)` to the Car class. As you can see in the next chunk of code, same name method can coexist

in the same class, as soon as they display a different arity, different kind of parameters. In this case, taxes can consider only the vanRate, as described in the first method version, `taxes(double vanRate)`

```
private double taxes( double vanRate) {  
    return (price-cost)*vanRate;  
}
```

However, we would like that our program take into consideration local taxes, that some regions or states apply to car sales. In order to offer this different functionality, the only thing we need to do is add this different version of the taxes method. Fortunately, the Java compiler is able to distinguish about this two method calls, owing that the number and/or the parameters type are different.

```
private double taxes( double vanRate, double localTaxes) {  
    return (price-cost)*vanRate + (price-cost)*localTaxes;  
}
```

`this.taxes(0.20)` method call would reference the first method version blue highlighted.

`this.taxes(0.16, 0.5)` method call would reference to the yellow highlighted.

Overloading is not strict or pure polymorphism because of the fact that the compiler know in compile Time which method we are referencing.

In Java, we can overload a method as many times as we need. In addition, we can do it for the constructor also, as depicted in the code below. Replace the previous version of Car constructor with these two new versions. In doing so, we are offering more possibilities to the programmer to create a car object.


```
public Car (String color, String brand, String model) {  
    this.color=color;  
    this.brand=brand;  
    this.model = model;  
}  
  
public Car (String color, String brand, String model, double price, double cost)  
{  
    this.color=color;  
    this.brand=brand;  
    this.model = model;  
    this.price=price;  
    this.cost=cost;  
  
    incrementNumberOfCarsCreated();  
}
```

Practice

It is also recommended in Java, for serialization purposes , to add a parameter-less version of your class constructors. You should start adding this version of the constructor to all your classes. Here an example for the class Car.

```
public Car () {  
  
}
```

Let us do it with all the classes in the model, SUV, SUVElectric and so on.

Overriding

Overriding means having two methods with the same method name and parameters

(i.e., *method signature*). One of the methods is in the parent class and the other is in the child class. Overriding allows a child class to provide a specific implementation of a method that is already provided its parent class.

To keep illustrating polymorphism, we are to explaining the overriding. As a result, we will change a little bit our subclasses. Firstly, we will modify the to string Method in our subclasses.

Add this method to SUV

```
@Override
    public String toString() {
        return "SUV [color=" + color + ", brand=" + this.getBrand() + ", model="
+ this.getModel() + "];"
    }
```

And this one to Sedan

```
@Override
    public String toString() {
        return "Sedan [color=" + color + ", brand=" + this.getBrand() + ",
model=" + this.getModel() + "];"
    }
```

The first thing you should take into consideration is the @Override tag. When we implemented a method that it is already implemented in the superclass, we need to add the @Override. We are overwriting a new code for this method, which is similar to the one in the superclass, owing that it receives the same parameters. In addition, the returned type is the same, String, and also the message is quite similar. Compare it with the original one:

```
@Override
    public String toString() {
        return "Car [color=" + color + ", brand=" + brand + ", model=" + model +
    "]"
    }
```

The idea is that each subclass can have a different behavior although all are similar. Responses for each class could be slightly different. We print each type of car differently.

Here are some important facts about Overriding and Overloading:

1. The real object type in the run-time, not the reference variable's type, determines which overridden method is used at *runtime*. In contrast, reference type determines which overloaded method will be used at *compile time*.
2. Polymorphism applies to overriding, not to overloading.
3. Overriding is a run-time concept while overloading is a compile-time concept.

To fully depict this difference we are set the next example. Add this code to your ApplInherit

```
Car car1 = new SUVElectric("red", "Tesla",  
                           "Model S High Performace", 50000, 30000,  
                           100, 17, 7);  
  
Car car2 = new Sedan("red", "BMW",  
                    "320", 50000, 30000);  
  
car1.toString();  
car2.toString();
```

In the next section, we will continue depicting the idea of polymorphism to abstract classes and subclasses.

3.5 Abstract Classes

Assuming that your hierarchy grows, and it is more specialize, you would not need to create Car class instances. Conversely, the idea is creating subclasses objects, such as SUV and Sport cars. As a result, the class Car becomes more a kind of blueprint or a template for subclasses. The abstract class is still implementing what is common to all subclasses, due to the fact that we avoid repeated code in OPP.

There are two interesting features of OPP and inheritance:

1. First one polymorphism we are already mentioned in the section above.
2. Another great feature or upside of inheritance is the capacity of a superclass to oblige to implement methods to subclasses, with the abstract modifier.

Let us get into work:

1. Add the abstract modifier to your class Car

```
public abstract class Car
```

Abstract classes are classes from which we cannot create objects. From now on, your program in MainClass should produce a compiler error if you try to create a Car object.

```
public abstract class Car
```

```
11 Car myTesla = new Car("red", "Tesla", "Model S High Performace", 50000, 30000);
12
13 System.out.println(myTesla);
14
15 Car myBMWi = new Car("blue", " BMW", "i5", 50000, 40000);
16
```

2. Change this lines to:

```
Car myTesla = new SUV("red", "Tesla", "Model S High Performace",
50000, 30000);

System.out.println(myTesla);

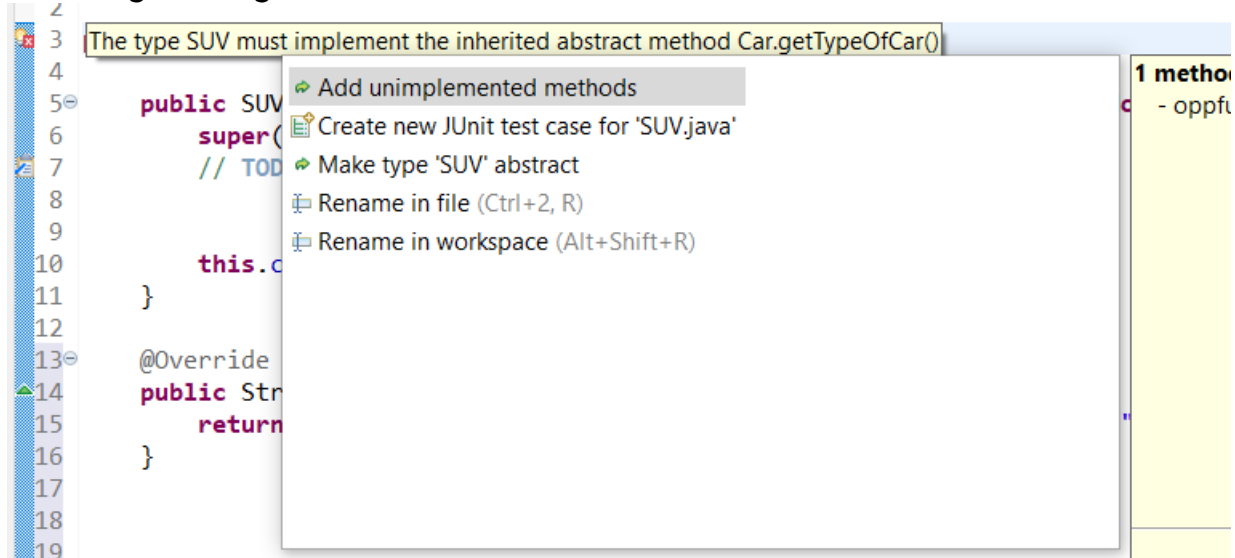
Car myBMWi = new Sedan("blue", " BMW", "i5", 50000, 40000);
```

3. And also this abstract method.

```
public abstract String getTypeOfCar();
```

Abstract methods are methods with no body, no implementation, we offer only the function signature. Their purpose is being implemented in subclasses which requires polymorphism.

The head to the SUV class. You will find also a compiler error. Don't be afraid. Eclipse will provide a solution. You are required to add the unimplemented methods. Click on Add unimplemented methods which results in the addition of the new abstract method `getTypeOfCar()` to your class. Having added the method, you should verify that this method has a body. Due to the fact that it is not abstract, you must implement it.



```
@Override
public String getTypeOfCar() {
    // TODO Auto-generated method stub
    return null;
}
```

Unfortunately, the method is returning null. You can replace this return statement with `return "SUV";`. Somehow, the use of literals in programming is not a good coding practice. I suggest that you should add a constant variable instead.

```
private static final String TYPE_SUV="SUV";
```

This modifications in your program should be alike the next excerpt of code:

```
package oppfundamentals;

public class SUV extends Car{

    private static final String TYPE_SUV="SUV";

    public SUV(String color, String brand, String model, double price, double cost) {
        super(color, brand, model, price, cost);
        // TODO Auto-generated constructor stub

        this.color= color;
    }

    @Override
    public String toString() {
```

```
        return "SUV [color=" + color + ", brand=" + this.getBrand() + ", model="
+ this.getModel() + "];"
    }

    @Override
    public String getTypeOfCar() {
        // TODO Auto-generated method stub
        return SUV;
    }

}
```

In the end, adding abstract methods compel subclasses to provide and implementation of them. What is remarkable here is that a superclass lead or impose the behavior of subclasses, owing that if they do not implement the abstract method a compiler error would bring up.

Practice

Following the last example, could you do the same for all classes?

3.6 Interfaces

In some Oriented Object Programming languages, subclasses can inherit from more than one class, i.e C++, which we label as multiple inheritance. Unfortunately, in Java multiple inheritance is not allow, and that results in subclasses extending from only one superclass. Nonetheless, Java offers a similar mechanism to partially simulate multiple inheritance, Interfaces.

Interfaces are similar to classes, but in opposition to classes, they do not offer code to be inherited by classes. Interfaces declare abstract methods that must be codify by classes that implements these interfaces. An interface is a collection of method names, without actual definitions, that indicate that a class has a set of behaviors in addition to the behaviors the class gets from its superclasses.

We can clearly distinguish in our model to kind of cars Electrics and Fossil Fuel Cars. Then the first thing we should do is separate both. Therefore, make USV Electric to inherit from Car as well.

```

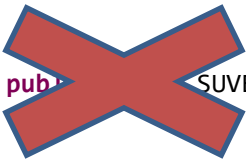
package oppfundamentals;

public class SUVElectric extends Car{
    private static final String TYPE_SUV_ELECTRIC="SUVElectric";
    public SUVElectric(String color, String brand, String model, double price, double
cost) {
        super(color, brand, model, price, cost);
    }

    @Override
    public String getTypeOfCar() {
        // TODO Auto-generated method stub
        return TYPE_SUV_ELECTRIC;
    }
}

```

In other OPP languages, to ameliorate the definition of electric we could have created a class Call Electric and extends it in SUVElectric. In Java it is not permitted.



```

public class SUVElectric extends Car, Electric { FORBIDDEN

```

To accomplish this desired model, we can use interfaces. We can define an Interface for Electric, labelled ElectricInterface, since we are interested in store, the battery Capacity and the Car Consumption per 100 km.

ElectricInterface.java

```

package oppfundamentals;

public interface ElectricInterface {

    public abstract double batteryCapacity ();

    public double powerConsumptionPerHundred ();

}

```

Having declare the methods, we can explain how interfaces performance. Look at the batteryCapacity method, and verify that it has been declared as abstract. Conversely, for the second one, we did not although it is not necessary. All methods declare but not define in Interfaces are abstract, yo do not need to use the abstract modifier. Hence, powerComsumptionPerHundred() is likewise abstract. **We do not use the abstract modifier in interfaces so that it is not needed.** Methods are abstract perse, unless we define their body.

Subsequently, we need to implement this interface in our SUV class. Moreover, it is mandatory to add this new information but only to electric cars.

Consequently, we need to:

1. Implementing the interface

```
public class SUVElectric extends Car implements ElectricInterface
```

2. Adding two new properties

```
private double bateryCapacity=0;  
private double compsumption=0;
```

3. Implementing the methods we were declared in the Interface

```
@Override  
public double batteryCapacity() {  
    // TODO Auto-generated method stub  
    return bateryCapacity;  
}
```

```
@Override  
public double powerComsumptionPerHundred() {  
    // TODO Auto-generated method stub  
    return powerComsumptionPerHundred();  
}
```



```
public class SUVElectric extends Car implements ElectricInterface {
    private static final String TYPE_SUV_ELECTRIC="SUVElectric";
    private double batteryCapacity=0;
    private double consumption=0;

    public SUVElectric(String color, String brand,
        String model, double price, double cost,
        double batteryCapacity, double consumption) {
        super(color, brand, model, price, cost);

        this.batteryCapacity=batteryCapacity;
        this.consumption= consumption;
    }

    @Override
    public String getTypeOfCar() {
        // TODO Auto-generated method stub
        return TYPE_SUV_ELECTRIC;
    }

    @Override
    public double batteryCapacity() {
        // TODO Auto-generated method stub
        return batteryCapacity;
    }

    @Override
    public double powerConsumptionPerHundred() {
        // TODO Auto-generated method stub
        return powerConsumptionPerHundred();
    }
}
```

It is assumed that when classes implement an Interface, they sign a contract with this interface. Resulting from this contract, classes must implement abstract interface methods. Similar to abstract classes, interfaces impose a behaviour to classes, obliging them to define, or implement methods. At this point, Electric cars must provide information about battery capacity and powerConsumption.

We can continue detailing even more our model. Imagine that you would like to manage information about car Seats, for Berlines and SUV. Then we can create a new Interface to accomplish your requirement, ISeats. We accustom in java to begin Interface names with capital I, following a word or multiword name that describe the interface function.

```
public interface ISeats {

    public int numberOfSeats();

}
```

Let's started adding changes to the SUVElectric. Whereas we can only extend from one class in Java, we can implement multiple interfaces. Highleighted in yellow, you can check changes for this new behavior, ISeats.

```
package oppfundamentals;

public class SUVElectric extends Car implements ElectricInterface, ISeats{
    private static final String TYPE_SUV_ELECTRIC="SUVElectric";
    private double bateryCapacity=0;
    private double compsumption=0;
    private int numberOfSeats=0;

    public SUVElectric(String color, String brand,
                        String model, double price, double cost,
                        double batteryCapacity, double compsumption,
                        int seats) {
        super(color, brand, model, price, cost);

        this.bateryCapacity=batteryCapacity;
        this.compsumption= compsumption;
        this.numberOfSeats=seats;
    }

    @Override
    public String toString() {
        return "SUV [color=" + color + ", brand=" + this.getBrand() +
            ", model=" + this.getModel() + ", bateryCapacity=" +
            batteryCapacity + ", compsumption=" + compsumption +
            ", numberOfSeats=" + numberOfSeats + "];"
    }

    @Override
    public String getTypeOfCar() {
```

```

        // TODO Auto-generated method stub
        return TYPE_SUV_ELECTRIC;
    }

```

```

@Override
public double batteryCapacity() {
    // TODO Auto-generated method stub
    return batteryCapacity;
}

```

```

@Override
public double powerConsumptionPerHundred() {
    // TODO Auto-generated method stub
    return powerConsumptionPerHundred();
}

```

```

@Override
public int numberOfSeats() {
    // TODO Auto-generated method stub
    return numberOfSeats;
}

```

```

}

```

Finally, we can modify our AppInheritance class to test our model changes

```

package oppfundamentals;

public class AppInheritance {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        SUVElectric myTeslaSUVElectric = new SUVElectric("red", "Tesla",
            "Model S High Performace", 50000, 30000,
            100, 17, 7);

        Car.numberOfCarsCreated();

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

```

```
        System.out.println("Total cars created: " + Car.numberOfCarsCreated());

        myTeslaSUVElectric.rePaint("White");

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

    }

}
```

Here the program execution outcome expected:

```
My new SUV:SUV [color=red, brand=Tesla, model=Model S High Performace,
batteryCapacity=100.0, compsumption=17.0, numberOfSeats=7]
Total cars created: 1
My new SUV repainted:SUV [color=White, brand=Tesla, model=Model S High Performace,
batteryCapacity=100.0, compsumption=17.0, numberOfSeats=7]
```

Practice

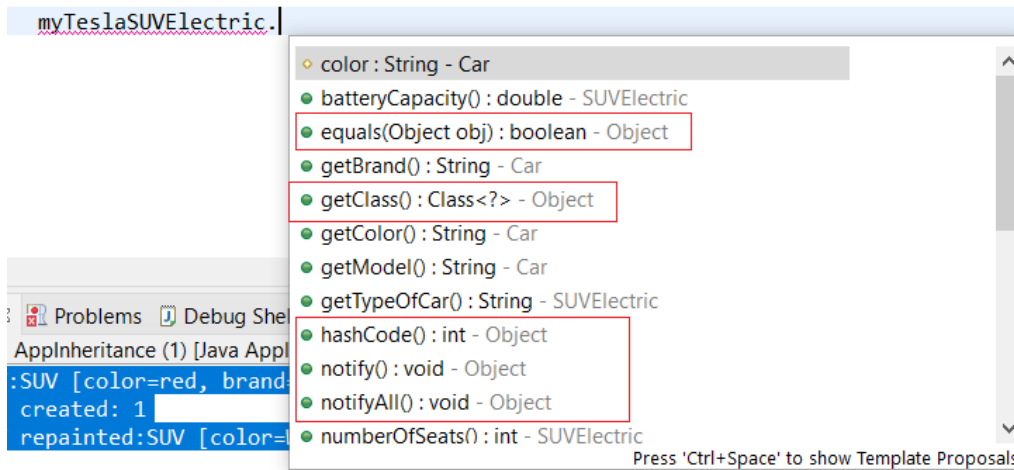
1. Include ISeat in SUV and Sedan class
2. Create a new new class SedanElectric, add ISeat and ElectricInterface contract to this class.
3. Create a new class SedanElectric, including the aforementioned interfaces.

Note: Notice how we massively use polymorphism in interfaces.

3.7 The Object Class in Java

From here to now, you have created a model of classes, including abstractions, encapsulation, inheritance, interfaces, polymorphism. After introducing the four pillars of OOP, designing and implementing a model, there is a question unanswered?

Would you are able to investigate a bit in your variables, there are some methods in your class USVElectric that you did not define:



Equals, getClass, hashCode, notify, notifyAll are not part of your code, but you can use them. Where do they come from?

Practice

1. Work in pairs, and search in the internet about these methods. Let me give you a hint. **All this methods are followed by Object.**
2. Elaborate an appropriate response.

3.8 The java Object class

We have study inheritance in this unit. In the previous exercise we have pointed out that there are some methods in our classes that we do not know about. How this is possible! The answer is simple. Java offers in his API a base class named Object. Any time you create a class, implicitly, your class inherits from Object. It means that Object is at top of any class model hierarchy, as we illustrate in the figure below.

This is applicable to any class that Java offers to programmers in the JAVA API we will introduce in the next chapter. You can also create object from the object class, using some of the methods provided by Object.

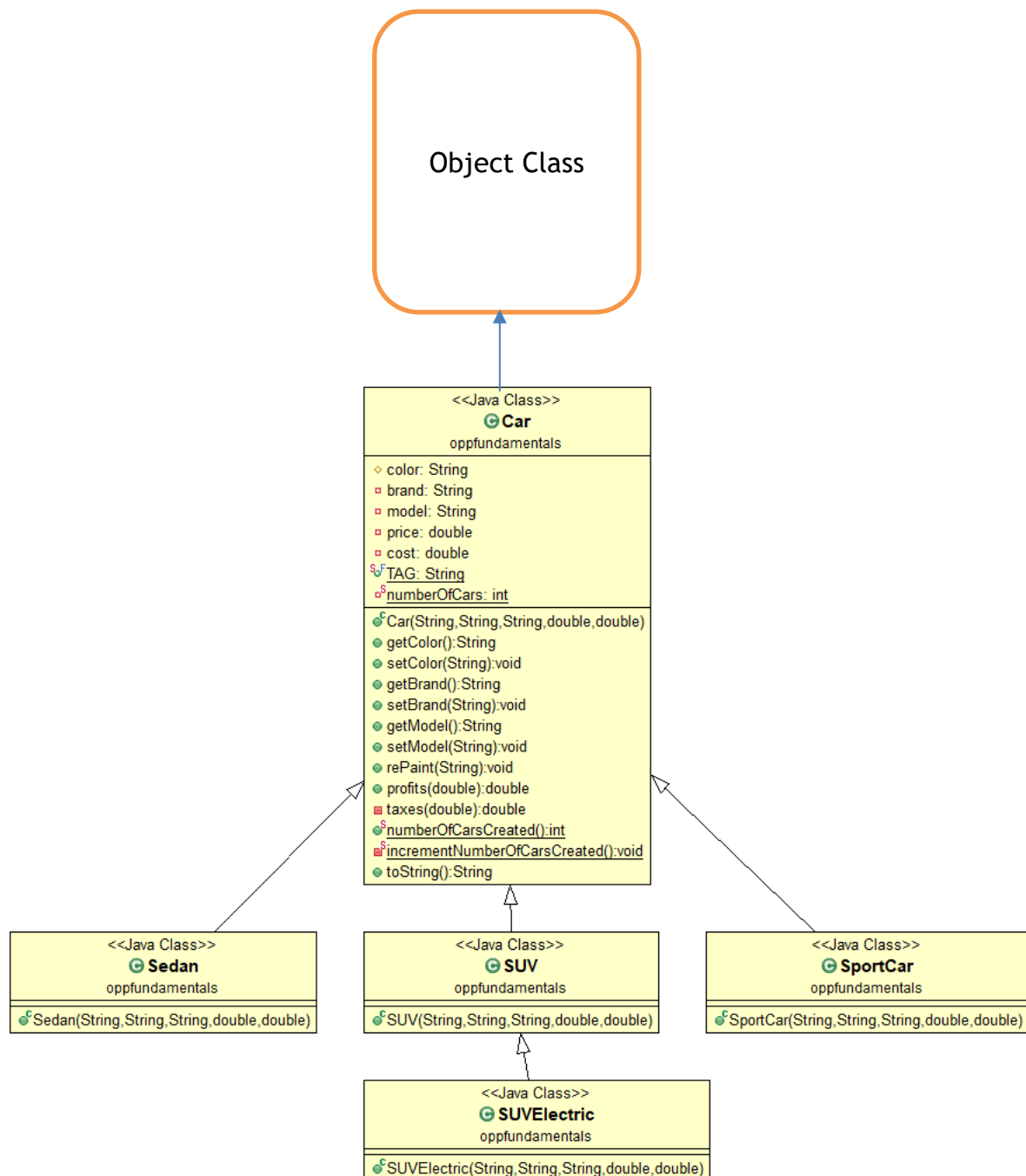
```
Object obj=getObject();//
```

Creating object of the Object class in Java is purposeless. The aim of Object is to be a template for your classes, and also offer and structure for inheritance. After the figure, you can find a table with all object class methods description. Two of these methods are extremely useful for Java programmers, equals and hasCode. We are to look over them and override them in our classes. To unravel the full process we will set some examples.

Equals is intended to compare objects from our classes, if we override this method in our

class. Owing that we are inherit this methods in our classes from Object, we are allowed to define or override them.

HashCode it is used in some Java Collections. Collections are a set or a list of objects. Java use hashCode to order or sort Objects in a collection. Also use equals, when to hashCode methods turn to be the same or return the same number. As a consequence, it is recommended to implement this methods throughout your classes if you will use them in a collection. Using object from our model in Collections is highly likely.



The Object class provides many methods. They are as follows:

Method	Description
<code>public final Class getClass()</code>	returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
<code>public int hashCode()</code>	returns the hashcode number for this object.
<code>public boolean equals(Object obj)</code>	compares the given object to this object.
<code>protected Object clone()</code> throws <code>CloneNotSupportedException</code>	creates and returns the exact copy (clone) of this object.
<code>public String toString()</code>	returns the string representation of this object.
<code>public final void notify()</code>	wakes up single thread, waiting on this object's monitor.
<code>public final void notifyAll()</code>	wakes up all the threads, waiting on this object's monitor.
<code>public final void wait(long timeout)</code> throws <code>InterruptedException</code>	causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes <code>notify()</code> or <code>notifyAll()</code> method).
<code>public final void wait(long timeout,int nanos)</code> throws <code>InterruptedException</code>	causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes <code>notify()</code> or <code>notifyAll()</code> method).
<code>public final void wait()</code> throws <code>InterruptedException</code>	causes the current thread to wait, until another thread notifies (invokes <code>notify()</code> or <code>notifyAll()</code> method).
<code>protected void finalize()</code> throws <code>Throwable</code>	is invoked by the garbage collector before object is being garbage collected.

3.8.1 Equals and Hascode.

We have already learned from the previous table the equals and the hashCode method. Hence, we can apply them in the next example. We will use these methods with the object class and the String class. But first, let us outline implementations of both method in object and String.

The equals method for object, returns true only if the object variables point to the same memory position. It means that any new object you create for the Object class is different. In the following example the result of executing these lines would be:

```
For OBJECT
are o1 and o2 equal? false
are o1 and o3 equal? true
```

```
Object o1= new Object();
Object o2 = new Object();

Object o3 = o1;

System.out.println("For OBJECT");

System.out.println("are o1 and o2 equal? " + o1.equals(o2));

System.out.println("are o1 and o3 equal? " + o1.equals(o3));
```

O1 and o2 are different so that any new type Object object you create is differente and unique. As you have assigned o1 to o3, Object o3 = o1; , they point to the same memory address, to the same object, thus they are equal.

The hashCode implementation of object, gives an identifier or unique long number to any new object you create. As you can check in the execution below, o1 and o3 hashCode is the same.

```
System.out.println("o1 hashCode " + o1.hashCode());
System.out.println("o2 hashCode " + o2.hashCode());
System.out.println("o3 hashCode " + o3.hashCode());
o1 hashCode 1101288798
o2 hashCode 942731712
```

String class offers and implementation, overrides, both methods. The equals method for String return true if both Strings has the same characters in the same order.

```
String s1 = new String("er");
String s2 = "er";

System.out.println("For STRING");

System.out.println("are bot strings, s1 y s2 equal?" + s1.equals(s2));

System.out.println("s1 hashcode " + s1.hashCode());
System.out.println("s2 hashcode " + s2.hashCode());
```

```
For STRING
are bot strings, s1 y s2 equal?true
s1 hashcode 3245
s2 hashcode 3245
toString de s1 er
toString de s2 er
```

Since s1 and s2 contain the same character “er” they are equal.

Receiving as an input the String characters, the String definition of hashCode generates a number based on these characters. The idea is that hashCode transforms each character to a byte, a number. Then, hashCode concat all this numbers.

The character “e” is a 32 if you convert it to byte. For “r”, the equivalente byte is 45. Consequently, the hashCode for “er” is 3245. At last, s1 and s2 has the same hashCode.

```
package objectclass;

public class equalsHashCodeExample {

    public static void main (String [] args) {

        Object o1= new Object();
        Object o2 = new Object();

        Object o3 = o1;
```

```
System.out.println("For OBJECT");

System.out.println("are o1 and o2 equal? " + o1.equals(o2));

System.out.println("are o1 and o3 equal? " + o1.equals(o3));

System.out.println("o1 hashCode " + o1.hashCode());
System.out.println("o2 hashCode " + o2.hashCode());
System.out.println("o3 hashCode " + o3.hashCode());

System.out.println("toString for o1" + o1.toString());
System.out.println("toString for o2" + o2.toString());


String s1 = new String("er");
String s2 = "er";

System.out.println("For STRING");

System.out.println("are bot strings, s1 y s2 equal?" + s1.equals(s2));

System.out.println("s1 hashCode " + s1.hashCode());
System.out.println("s2 hashCode " + s2.hashCode());

System.out.println("toString de s1 " + s1.toString());
System.out.println("toString de s2 " + s2.toString());

}

}
```

3.8.2 Overriding hashCode and Equals in our classes

You will try to introduce overriding in the equals method in SUVElectric, therforth, you may copy this code to your SUVElectric car. Supposing that two SUVElectric are equals if and only if the objects are not null and battery capacity, number of seats, model, color and brand properties are the same.

```
@Override
public boolean equals(Object obj) {

    if (obj == null)
```

```

        return false;
    if (getClass() != obj.getClass())
        return false;
    SUVElectric suv = (SUVElectric) obj;
    if (batteryCapacity == suv.batteryCapacity &&
        numberOfSeats == suv.numberOfSeats &&
        this.getModel().equals(suv.getModel()) &&
        this.getColor().equals(suv.getColor()) &&
        this.getBrand().equals(suv.getBrand()) )

        return true;

    else
        return false;
}

```

For the hashCode method we are going to take advantage of the hashCode String method to generate a hashCode. We can concat in a String battery capacity, number of seats, model, color and brand. After that, we should return the hashCode of this String:

```

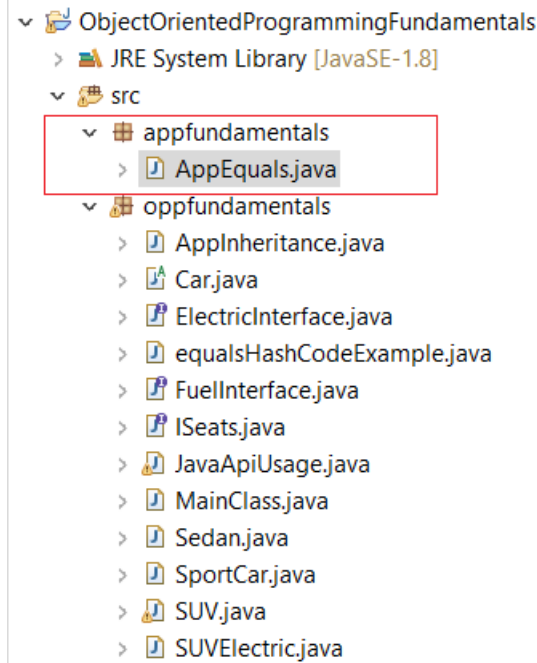
@Override
public int hashCode() {

    String stringOfFields = batteryCapacity + this.batteryCapacity +
        this.numberOfSeats +
        this.getModel() +
        this.getColor() +
        this.getBrand();

    return stringOfFields.hashCode();
}

```

Providing that you have already add these methods, let us have a try. To do so, we will create a new package, appfundamentals, and a new class, AppEquals.java



And add this code to the AppEquals.java class

```
package appfundamentals;
```

```
import oppfundamentals.SUVElectric;
```

```
public class AppEquals {
```

```
    public static void main(String[] args) {
        // TODO Auto-generated method stub
```

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

```
        SUVElectric myTeslaSUVElectric2 = new SUVElectric("red", "Tesla",
            "Model S High Performace", 55000, 32000,
            100, 17, 7);
```

```
        System.out.println("Are both cars equal? " +
            myTeslaSUVElectric.equals(myTeslaSUVElectric2));
```

```
        System.out.println("myTesla hashCode " + myTeslaSUVElectric.hashCode());
        System.out.println("myTesla2 hashCode " + myTeslaSUVElectric2.hashCode());
```

```
    }
```

```
}
```

The result of the executions should look alike this:

```
Are both cars equal? true
myTesla hashCode -812946937
myTesla2 hashCode -812946937
```

Notice that both cars have different cost and price. Anyway, they are equals resulting from the fact that we have not considered price and cost relevant to our equal method. Likewise, have a look at this sentence. We will sketch it in detail in the next section.

```
import oppfundamentals.SUVElectric;
```

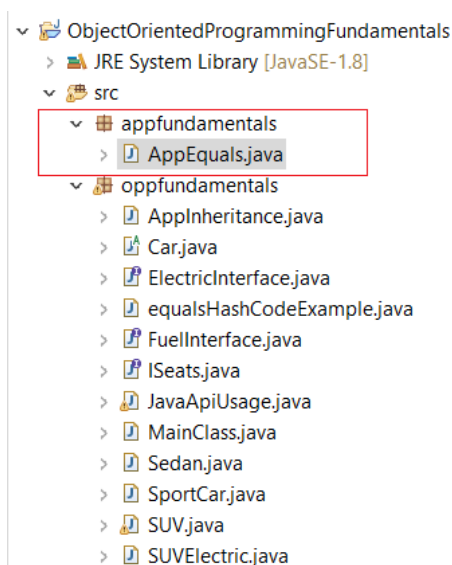
IMPORTANT NOTE: Henceforward, any time we provide examples the student must guess which packages should create to add the class to his project. Heed your attention again to the preceding example.

```
package appfundamentals;
```

```
import oppfundamentals.SUVElectric;
```

```
public class AppEquals {
```

The package for this class is appfundamentals. From now on, the student must verify the package for each example, and create it before adding the class.



1. Override equals and hashCode in SedanElectric, following the same pattern we have used for SUVElectric.
2. Create to SedanElectric in AppEquals. Compare them and display this comparison and hashCodes in the console.

3.9 Enum types in Java

Enumerations serve the purpose of representing a group of named constants in a programming language. For example the 4 suits in a deck of playing cards may be 4 enumerators named Club, Diamond, Heart, and Spade, belonging to an enumerated type named Suit. Other examples include natural enumerated types (like the planets, days of the week, colors, directions, etc.).

Enums are used when we know all possible values at compile time, such as choices on a menu, rounding modes, command line flags, etc. It is not necessary that the set of constants in an enum type stay fixed for all time.

In Java (from 1.5), enums are represented using the enum data type. Java enums are more powerful than C/C++ enums . In Java, we can also add variables, methods and constructors to it. The main objective of enums is to define our own data types(Enumerated Data Types).

They have the form or the structure of a class although they are not a class. They can have a constructor and they have some predefined methods we will study in this section. To declare enumerates we follow the next form, similar to a class.

```
Enum Type_name {  
  
value1, value2, value3 .....  
  
}
```

For example:

```
enum Color  
{  
    RED, GREEN, BLUE;  
}
```

This is the simplest version of enumerates. Java offers more complex structure we will

sketch. The use of this constants is similar to the use of the static properties in a class. We write the enum name, dot, and the name of the constant or property. We have include the enum in a class file, despite of the fact that it is recommended to create a file for each enumerate type we add to our program. We will do it in the next section.

```
Type_Name.PROPERTY_NAME;
```

```
Color.RED;
```

We can create variables of this enum Types similarly to any other type;

```
Color c1 = Color.RED;
```

```
package enumerates;
```

```
//Declaration of enum in java :  
  
//Enum declaration can be done outside a Class or inside a Class but not  
inside a Method.  
  
// A simple enum example where enum is declared  
// outside any class (Note enum keyword instead of  
// class keyword)  
enum Color  
{  
    RED, GREEN, BLUE;  
}  
  
public class EnumExampleColor {  
  
    public static void main(String args[])    {  
        Color c1 = Color.RED;  
        Color c2 = Color.BLUE;  
        System.out.println(c1);  
    }  
}
```

Internally the enumerate type are implemented as classes. Therefore, the enum Color is implemented internally by the java compiler as the class Color . As the color class does not have the equals method overridden, each object color is different from the rest owing that it inherits from the object class, and we should recollect that each Type Object object we create is unique. As a consequence, any color it is create with the new Color() statement is different.

Important points of enum :

- Every enum internally implemented by using Class.

- `/* internally above enum Color is converted to`
- `class Color`
- `{`
- `public static final Color RED = new Color();`
- `public static final Color BLUE = new Color();`
- `public static final Color GREEN = new Color();`
- `*/`

Enum and Inheritance :

All enums implicitly extend `java.lang.Enum` class. As a class can only extend one parent in Java, so an enum cannot extend anything else.

- **`toString()`** method is overridden in `java.lang.Enum` class, which returns enum constant name.

enum can implement many interfaces.

- **`values()`, `ordinal()` and `valueOf()`** methods :

These methods are present inside `java.lang.Enum`.

- **`values()`** method can be used to return all values present inside enum.
- Order is important in enums. By using **`ordinal()`** method, each enum constant index can be found, just like array index.
- **`valueOf()`** method returns the enum constant of the specified string value, if exists

We can change the main function of the prior example:

```
public static void main(String args[]) {
    Color c1 = Color.RED;
    Color c2 = Color.BLUE;

    Color c3= Color.valueOf("GREEN");
    System.out.println(c1);

    System.out.println(c1 + " color order is " + c1.ordinal());
    System.out.println(c2 + " color order is " + c2.ordinal());
    System.out.println(c3 + " color order is " + c3.ordinal());

    if (!c1.equals(c2)) {
        System.out.println(c1 + " color and " + c2 + " color are
different");
    }
}
```

```
    }  
    }  
}
```

```
RED  
RED color order is 0  
BLUE color order is 2
```

RED color and BLUE color are different

From this execution we can infer:

1. Red color ordinal, order number is 0

```
Color c1 = Color.RED;  
c1.ordinal();
```

2. RED and BLUE are different colors:

```
if (!c1.equals(c2))
```

3. You may convert from String to Color using the valueOf method.

```
Color c3= Color.valueOf("GREEN");
```

Note: Try this conversion `Color.valueOf("Green");` to check if the method is case insensitive

3.9.1 More complex enumerate types.

The consequences of enum types being implemented as classes is that we can define more complex structures for this enum Types. To illustrate so, we will set and deconstruct the following example. We will include the enum type in its own file. Hence, add a new enum type to your project, as it is depicted in the picture below



Call the file `ComplexEnumerateStudent.java` and copy the subsequent code in it. Enum type files are identical to class type files. We must add the public modifier as we did for classes `public enum ComplexEnumerateStudent.`

This new enum is more complex since we have parameters for each enum element. In this case we have three parameters, String, int, String. When we define these kinds of enum, it is mandatory to add a Constructor which consider and include these parameters.

```
EXCHANGE_STUDENT("EXCHANGE STUDENT", 1, "INTERNATIONAL"),
```

```
ComplexEnumerateStudent(String description, int id, String name) {
```

Likewise, It is suggested that properties are included in the file, and assigned into the constructor method.

```
private String description;
private int id;
private String name;
```

Provided that you have properties, you have the ability to add getters to the enum type, to retrieve these properties in your program if necessary.

```
public String getDescription() {
```

```
    return this.description;
}
```

```
public int getId() {
    return id;
}
```

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

ComplexEnumerateStudent.java

```
package enumerates;  
  
public enum ComplexEnumerateStudent {  
  
    EXCHANGE_STUDENT("EXCHANGE STUDENT", 1, "INTERNATIONAL"),  
    NATIONAL_STUDENT("NATIONAL STUDENT", 2, "NATIONAL"),  
    STATE_STUDENT("STATE STUDENT", 3, "LOCAL");  
  
    private String description;  
    private int id;  
    private String name;  
  
    ComplexEnumerateStudent(String description, int id, String name) {  
        // TODO Auto-generated constructor stub  
        this.description=description;  
        this.id=id;  
        this.name=name;  
    }  
  
    public String getDescription() {  
  
        return this.description;  
    }  
  
    public int getId() {  
        return id;  
    }  
  
    public String getName() {  
        return name;  
    }  
}
```

}

After declare and define the enum, we can use it in a new program

```
package enumerates.app;

import enumerates.ComplexEnumerateStudent;

public class AppEnumerates {

    public static void main(String args[]) {

        ComplexEnumerateStudent studentType1 =
ComplexEnumerateStudent.EXCHANGE_STUDENT;
        ComplexEnumerateStudent studentType2 =
ComplexEnumerateStudent.STATE_STUDENT;

        ComplexEnumerateStudent studentType3 =
ComplexEnumerateStudent.valueOf("NATIONAL_STUDENT");

        System.out.println("Properties for student " + studentType1 + " are: ");
        System.out.println("Description: " + studentType1.getDescription() +
            " Id: " + studentType1.getId()
            +" Name: " + studentType1.getName());

        System.out.println("Ordinal for student " + studentType3+ " "+
studentType3.ordinal());

    }

}
```

In blue, It is highlighted the methods inherited from enum. We are using `valueOf` or `ordinal()` the same manner we did in the `enum Color` example. In addition, we are calling to the getters we have defined for this new enum, i.e `studentType1.getDescription()`.

Activity.

Create your own enum for Type of cars EnumCars. According to the description below, you should codify: the Enum, properties, constructor and getters.

Types:

ELECTRIC: 1, "Electric Engine", "SEV"
DIESEL: 2, "Diesel Engine, "DV"
GAS: 3, "Gas Engine, "GV"
HYDROGEN: 4, "Hydrogen Engine", "HEV"

The file name will be `TypeCars.java`.

4 Property, Variable, and parameter scope and passing parameters techniques

Once we have depicted the principal Java characteristics, it urges to address some details that are gist in our programming algorithms in Java. Firtly, we should widen the concept of variable declarations and their scope. Secondly, we will study the effects of passing parameters to methods.

4.1 *Variable scope*

Having said that variable declaration is a common assignment in Java, programmers can attain different effects since they can declare variables in different bodys of statement blocks. Generally, a variable scope is restrain by the block of code within it is declared. Hence:

If we declare a variable in the main program block statement, the scope of this variable is the main program. That is, the variable can be referenced anywhere in the main statement block.

In the next example, the `variableScopeMain` scope is the public static main method. Observe how we can use the `variableScopeMain` in any block statement in the program. Otherwise, the `variableScopeIf` scope is the if sentence block statement. In the event of your trying to assign the `variableScopeIf` out of the if statement, the compiler would show an error. Consequently, you should declare variables considering places where they will be utilized.

In regard to parameters, we could say the same. The scope of a parameter in Java is the method block statement which belongs to. We could use the `args` parameter anywhere in the main method.

```
// main method
public static void main(String[] args)
{
    int variableScopeMain =0;

    if (variableScopeMain >-1) {
        int variableScopeIf=0;
        variableScopeMain =1;

        variableScopeIf = variableScopeMain;
        System.out.println("Parameters " + args);
    }
    // variableScopeIf =3;

    for (int i = 0; i<3 ; i++) {
        int variableScopeFor = 3 + variableScopeMain;
        // variableScopeFor = 3 + variableScopeIf;

        System.out.println("Parameters " + args);
    }
}
```

To sum up, variables are always local in Java. Let us review object properties and their modifier. **The scope of a property is global to the object.** We can use properties anywhere in the class, methods, or block statements within the methods.

If we look back to our class Car example, we could use any **private property** anywhere in the class code. Nevertheless, we could not reference a private property outside the class code, even in subclasses of car, such as Sedan. Conversely, the protect String color property, can be use in the class code, and can be reference in any subclass that extends from Car, directly or indirectly.

```
public abstract class Car {
    protected String color;
    private String brand;
    private String model;
    private double price;
    private double cost;

    public static final String TAG = "Car Object";
    private static int numberOfCars = 0;

    public abstract String getTypeOfCar();

    public Car() {
        incrementNumberOfCarsCreated();
    }

    public Car (String color, String brand, String model) {
        this.color=color;
        this.brand=brand;
        this.model = model;
        incrementNumberOfCarsCreated();
    }
}
```

Finally, the public modifier let us reference the property anywhere in our code where we create an object of this class. In the next program, three properties have been declared:

```
private int myProperty =3;

public int myPropPublic =3;

public static int staticPublicProperty=4;
```

We can reference them anywhere in the class code.

```
public int miFunction(int num) {
```



```
int num2=2 + myProperty + myPropPublic + staticPublicProperty;
```

On the contrary, we cannot use the private myProperty in the main method.

```
ExampleVariableScope ex = new ExampleVariableScope();
//ex.myProperty;
ex.myPropPublic=4;
ExampleVariableScope.staticPublicProperty=5;
```

However, we can make use of the public property ex.myPropPublic through the object ex. We can also reference the class public property ExampleVariableScope.staticPublicProperty through the class name.

ExampleVariableScope.java

```
public class ExampleVariableScope {

    private int myProperty =3;
    public int myPropPublic =3;
    public static int staticPublicProperty=4;

    public int miFunction(int num) {

        int num2=2 + myProperty + myPropPublic + staticPublicProperty;

        return num2 + num;

    }

    // m method
    public static void main(String[] args)
    {

        int variableScopeMain =0;

        if (variableScopeMain >-1) {

            int variableScopeIf=0;
            variableScopeMain =1;

            variableScopeIf = variableScopeMain;
            System.out.println("Parameters " + args);

        }
        // variableScopeIf =3;

        for (int i = 0; i<3 ; i++) {
```

```
        int variableScopeFor = 3 + variableScopeMain;
        // variableScopeFor = 3 + variableScopeIf;

        System.out.println("Parameters " + args);

    }

    ExampleVariableScope ex = new ExampleVariableScope();
    //ex.myProperty;
    ex.myPropPublic=4;
    ExampleVariableScope.staticPublicProperty=5;

}

}
```

Practice

Create a new class `ClassReferenciaPropiedades`. In this class constructor create an object of the `ExampleVariableScope`. Try to reference the three properties. Explain with your own words the results.

4.1.1 Passing Parameter techniques in Java

Passing parameters strategies have been used in programming even before the born of Java. Traditionally, there are two ways to pass parameters in C: Pass by Value, Pass by Reference.

- Pass by Value. Pass by Value, means that a copy of the data is made and stored by way of the name of the parameter. ...
- Pass by Reference. A reference parameter "refers" to the original data in the calling function.

As we can use both in Java, we will try to explain these techniques through Java examples

In the next example, we have built a class `PersonPassinParameters`, that has three properties `name`, `lastName` and `age`. It also has the getters and setters and the `toString` method. Finally we have also add two more methods to explain passing parameters, `parameterByValueAge` and `parameterAsReferenceCapitalize`.

In the main program we have to local variables:

```
int newAge =45;
PersonPassinParameters person1= new PersonPassinParameters("johnny", "shotgun", 40);
```

As aforementioned, passing by value, after the function call, as soon as the function gain control of the execution flow, a copy of the passed parameter value is stored in the parameter. The value of `newAge` is 45.

```
person1.parameterByValueAge(newAge);
```

Given that the function is called, a copy of `newAge` value, 45 is store in the `age` parameter. Then, `age` increments by one in the function, then its value is 46. Here, it is printed in the console:

```
Age parameter value:46
```

```
public void parameterByValueAge(int age) {
    age = age+1;
    System.out.println("Age parameter value after modified:" + age);
    this.setAge(age);
}
```

Eventually, the functions ends its execution and the flow goes back to the main program.

```
System.out.println("Age variable after passing by value:" + newAge);
```

The result:

```
newAge variable after passing by value:45
```

Even though the parameter age has been modified, the newAge variable do not. This is a consequence of passing parameters by value, the parameter age has a copy of the variable newAge; However, the parameter does not have the variable itself, the address in memory were the data is stored. You can modify the parameter, but the variable passed as a parameter, remains the same. All primitive types in Java are passed by value since Java make copies of them if they function as passed values to a function call.

On the contrary, for objects in Java, complex types, the demeanor or behaviour is different. Remember that an object variable stores the object memory address. Considering the following call `parameterAsReferenceCapitalize`, we are passing as a parameter an object variable, which is a memory address.

```
person1.parameterAsReferenceCapitalize(person1);
```

```
System.out.println("Person1 after passed by reference:" + person1 );
```

The value of the `person1` variable, that is a memory address, is copied in the `person` parameter. Hence, both variable and parameter reference the same memory area. This memory address point to the object data as we have seen before.

```
public void parameterAsReferenceCapitalize(PersonPassinParameters person) {  
  
    person.setName(person.getName().substring(0,1).toUpperCase()  
        + person.getName().substring(1,person.getName().length()));  
  
    person.setLastName(person.getLastName().substring(0,1).toUpperCase()  
        +  
person.getLastName().substring(1,person.getLastName().length()));  
  
}
```

Given that both variable and parameter have the same value, a memory address 1000000, both accede to the same object data. It means if we modify the object through the parameter, that will be reflected in the object data, since the parameter does not contain the object data, but contains a copy of the memory address which points to the object. Consequently, when the method finishes, and the flow goes back to the main program, the object data has been modified, and It can be verified by the `person1` variable so that the variable can accede to the same object data. In the next figures, we can verify the effects of the method `parameterAsReferenceCapitalize` execution in memory.

Before the method

Memory address		value
400000	person1	1000000
..		
..		
..		
60000	Method Parameter asRefer....	
60000	Person parameter	1000000
1000000	Object Person	
	name	johny
	lastName	shotgun
	age	46

Copy of the person1 value

After the method

Memory address		value
400000	person1	1000000
..		
..		
..		
60000	Method Parameter asRefer....	
60000	person parameter	1000000
1000000	Object Person	
	name	Johny
	lastName	Shotgun
	age	46

After the method execution the object data is modified.

We can validate this statement following the program execution output:

This is the value of person1 object before calling the method
parameterAsReferenceCapitalize

Person1 before passed by reference: PersonPassinParameters [name=johny, lastName=shotgun, age=46]

After calling the method, the object has been modified. First name and lastname has been capitalized.

Person1 after passed by reference: PersonPassinParameters [name=Johny, lastName=Shotgun, age=46]

Overall, the main point of passing parameters by reference is that the parameter copies a reference that points to (a memory address) to the real data. Containing a reference to the real data, the parameter is able to modify this data. As soon as the method ends, changes remain in the object data, and they can be verified in the main program.

On the other hand, in the passing parameters by value strategy, the parameter work with a copy of the data. If the parameter changes, a copy of the data is modified, not the original one. The variable in the main program is not affected, and the data does not experience any update.

PersonPassinParameters.java

```
package variablescopepassingparameters;

public class PersonPassinParameters {

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

    public PersonPassinParameters() {

    }

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

    public String getName() {
```

```
        return name;
    }
```

```
public void setName(String name) {
    this.name = name;
}
```

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

```
public void setLastName(String lastName) {
    this.lastName = lastName;
}
```

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

```
public void setAge(int age) {
    this.age = age;
}
```

```
public void parameterAsReferenceCapitalize(PersonPassinParameters person) {

    person.setName(person.getName().substring(0,1).toUpperCase()
        + person.getName().substring(1,person.getName().length()));

    person.setLastName(person.getLastName().substring(0,1).toUpperCase()
        +
person.getLastName().substring(1,person.getLastName().length()));

}
```

```
public void parameterByValueAge(int age) {

    age = age+1;

    System.out.println("Age parameter value after modified:" + age);
}
```



```
        this.setAge(age);
    }

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

    // m method
    public static void main(String[] args)
    {
        int newAge =45;
        PersonPassinParameters person1= new PersonPassinParameters("johny", "shotgun",
40);

        person1.parameterByValueAge(newAge);

        System.out.println("Age variable after passing by value:" + newAge);

        System.out.println("Person1 before passed by reference:" + person1 );
        person1.parameterAsReferenceCapitalize(person1);

        System.out.println("Person1 after passed by reference:" + person1 );

    }
}
```

Activity

Create a class `MyActivityParameters`. Add a main function to it. Implements this method `public static void checkParameter(int i)` modifying the `i` parameter.

Call it in the main program:

```
int var = 5;
```

```
checkParameter(var). Will be the variable var modified?
```

Now

```
public static void checkParameter(Integer i) modifying the i parameter.
```

```
Integer var2 = 5;
```

```
checkParameter(var2). Will be the variable var modified?
```

5 The java APIs.

The **Java API** is an Application Programming Interface (API) provided by the creators of the Java programming language, which provides developers with means to develop Java applications.

Because the Java language is an object-oriented language, the Java API provides a set of utilitarian classes to perform all kinds of tasks required within a program. The Java API is organized into logical packages, where each package contains a set of semantically related classes.

There are numerous Java APIs to perform all kinds of operations, 1 some of the best known are:

- **JAXP**- To process XML.
- **Servlets**- to facilitate the implementation of web solutions.
- **Hibernate**- To facilitate persistence deployment.

The function of java APIs is intended to provide developers with tools to build applications. They make our lives easier, allowing us to reuse code, rather than having to

program it ourselves. Tasks such as, having classified data, **doing mathematical operations**, **doing text searches**, asking **the user** for data, and many more processes are issues that are frequently repeated in programming, and therefore will be resolved in the Java API. Of course, we can **create our own algorithms to sort lists**, but **the fastest and most efficient thing, in general, will be to use the API tools** available because they are developed by professionals and have been debugged and optimized over the years, and over versions of the language.

5.1.1 Java.base, java.lang y javax

Java.base defines the fundamental APIs of the java SE Platform, and therefore the basis for **developing applications that run on top of the JRE**, which contains the JVM. Providers: **The JDK is a module that provides an implementation of the jrt file system provider to enumerate and read class and resource files in a runtime image.** It is the one that allows us **to create and manage files of classes and resources, compile, execute our programs.** It is the basis on which more APIs and applications are developed.

java.lang. It provides **classes that are fundamental to the design of the Java programming language.** The most important classes are **Object**, which is the root of the class hierarchy, and **Class**, instances of which represent classes at run time. It allows us **to create new classes and objects, inheritance and polymorphism.**

These **two libraries do not need to be imported**, nor required in modules, they are **imported implicitly.** In any case, **all java libraries are organized in packages just as we must organize our components of our applications into packages.** For example, for the file library, the **documentation will refer to it as api java.io or package java.io.**

The Java programming language uses the **javax prefix for a standard Java extension package.** These include **extensions such as javax.servlet**, which deals with running **servlets.** The **jcr API**, which deals with the **Java content library** and access to **repositories such as CMS.** The **swing API** that deals with the **writer interfaces**, etc. In short, **it is used for extensions such as Java Enterprise, development for enterprise applications.**

- **Basic java libraries, expanding knowledge**

When we have experience as Java **programmers**, we may have **classes developed by ourselves** that we use in different projects. In **large**, it is common to **have classes developed by colleagues** of the company that we will use in a similar way to how the Java API is used: **knowing its interface but not its implementation.** Working with **a class without seeing its source code** requires that there be a **good documentation** that will guide us. We'll talk about **documentation of classes and projects in Java** a little later. For now, let's learn how to use the **Java API documentation.**

First, we need to **get an idea of how API classes are organized**. This organization is in the **form of a hierarchical tree**, as seen in the figure "Guidance scheme of the organization of libraries in the Java API". This **figure tries to show the Java API organization, but it does not list all existing packages or classes** that are many more and would not fit in one or more sheets.

Library names respond to this hierarchical scheme and are based on dot notation. For example, the **fully qualified name for the ArrayList class would be java.util.ArrayList**. The use of ***** is **allowed to name a set of classes**. For example, **java.util.*** refers to the set of **classes within the java.util package**, where we have ArrayList, LinkedList, and other classes.

To use API libraries, there are two situations:

- a) There are **libraries or classes that are always used** because they are fundamental elements of the Java language such as the **String class**. This class, belonging to the **java.lang package**, can be used directly in any Java program as it is loaded **automatically**.
- b) There are **libraries or classes that are not always used**. To use them within our code we must indicate that we **require their payload by means of an import statement** included in the class header. For example **import java.util.ArrayList**; it is a statement that is included in the header of a class that allows us to use the ArrayList class of the Java API. Writing **import java.util.***; would allow us to load all classes from the **java.util package**. Some packages have **dozens or hundreds of classes**. That's why we'd **generally prefer to specify classes** over using asterisks because it prevents the in-memory loading of classes that we're not going to use. An imported class can be used in the same way as if it were a **class generated by us**- we can create objects of that class and **call methods** to operate on those objects. In addition, each class will have one or **more constructors**.

Finding a **list of most used libraries or classes** is an almost impossible task. Each programmer, depending on their activity, uses certain libraries that may not be used by other programmers. Desktop programming-centric programmers will use classes other than those used by web or database management developers. Basic classes and libraries should be known through Java training courses or lessons. The most **advanced classes and libraries you will need to use and study them** as they become necessary for the development of **applications**, since their complete study is practically impossible. We can cite broad-use classes.

In the **package java.io**: File classes, Filewriter, FileReader, etc. In the **java.lang package**: System, String, Thread classes, andsoon. In the **java.security package** -Classes that allow you to implement encryption and security. In the **java.util package** package: ArrayList, LinkedList, HashMap, HashSet, TreeSet, Date, Calendar, StringTokenizer, Random, andsoon. In the **java.awt and javax.swing packages** a graphical library: development of **graphical user interfaces** with windows, buttons, etc.

We insist on one idea: **don't try to memorize the detailed java API organization or a list of commonly used classes** because this **makes little sense**. The important thing is that **you know how to organize, how classes are structured and used, and learn how to find information to find it quickly when needed**.

To program in Java, we will have to continually consult the Java API documentation. This **documentation is available on cds of specialized books and magazines or on the internet** by typing in a search engine such as yahoo, google or bing the text "api java 8" (or "api java 6", "api java 10", etc.) depending on the version you are using. The **Java API documentation in general is correct and complete**. However, in exceptional cases it may be incomplete or contain errors.

Many of the **documentation can be found on Oracle's official website**. To document THE APIs, the Javadoc application of the **JDK is commonly used that allows us to generate documentation from tags, comments and text files**, automatically generates the documentation.

Oracle's official site for java is:
<https://www.oracle.com/java/>

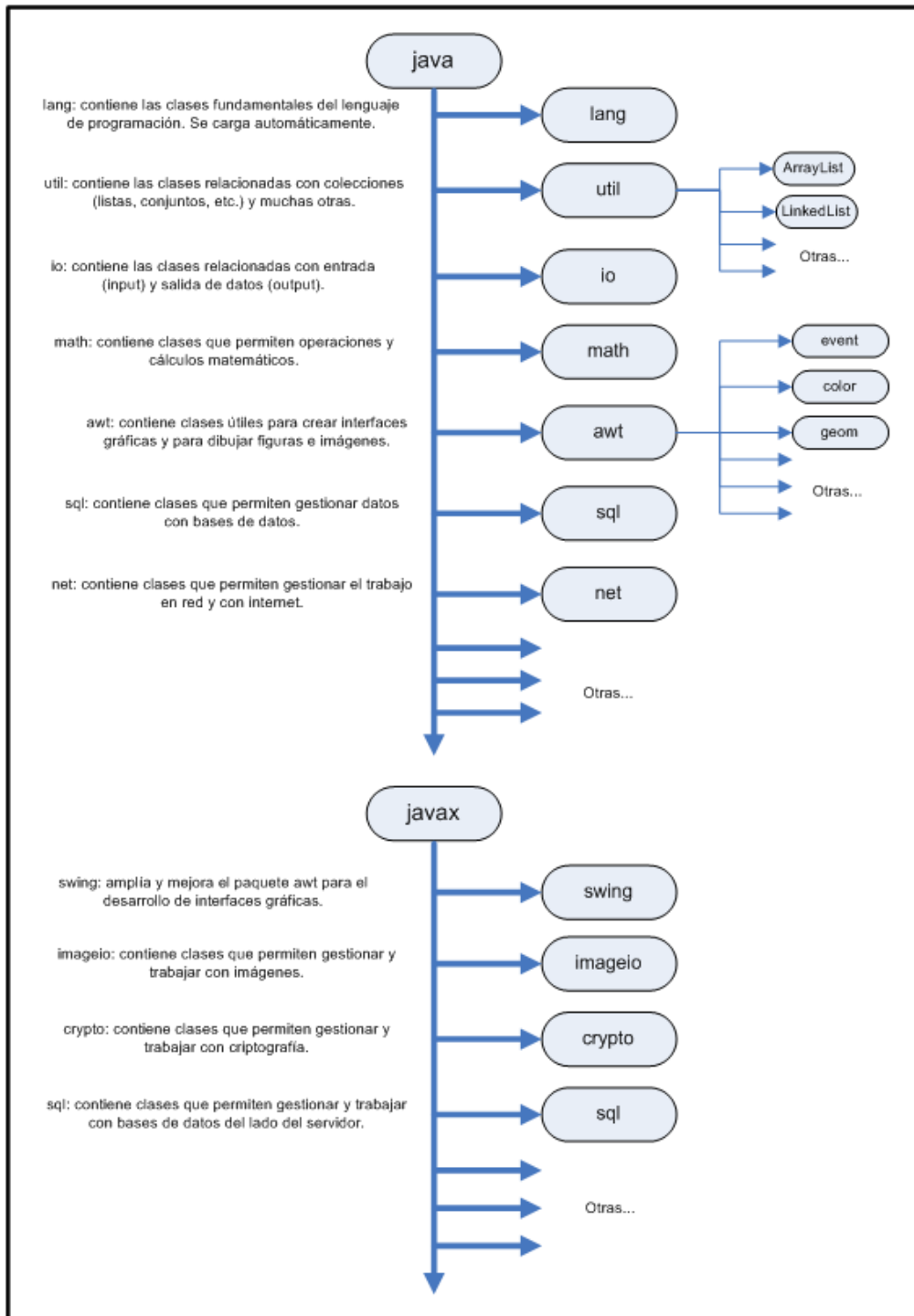
You can find **information about downloads, API's and more**. It is **our first source of documentation as programmers**, and it is mandatory to consult, as it is the official documentation. **Java JSE is the official version of Java developed by Oracle**, which offers us, among other things, the **JRE and the JDK**, Oracle's Java development kit

<https://docs.oracle.com/en/java/javase/11/docs/api/index.html>
<https://www.oracle.com/java/technologies/javase-downloads.html>

You will find **extra information at openJDK**. It is an **open platform for java developers**. It offers an **alternative implementation to the official version of java** offered in the JSE.

<https://openjdk.java.net/>

In the following image you have a graphic summary of the most used libraries.



5.2 Get started with the JAVA API

We fully explained the JAVA API, thus is time to make use of it. There is a powerful tool in Java programs that lets programmers use all packages, libraries, and eventually class provision. It is a reserved keyword called `import`. With that `import`, you can include these libraries and classes in your applications for usage purposes.

Built-in Packages

The Java API is a library of prewritten classes, that are free to use, and included in the Java Development Environment.

The library contains components for managing input, database programming, and much much more. The complete list can be found at Oracles website:

<https://docs.oracle.com/en/java/javase/14/docs/api/index.html>

The library is divided into **packages** and **classes**. Meaning you can either import a single class (along with its methods and attributes), or a whole package that contain all the classes that belong to the specified package.

To use a class or a package from the library, you need to use the `import` keyword:

There are some libraries in Java not need it to be imported, `java.lang`. These libraries are included in your program implicitly. Therefore, if you need to use one of this classes, the only thing you need to do is writing a variable for this class. For instance, `String` is a class that belongs to `java.lang`, to employ it, the only thing you must do is declaring a variable.

You can verify the former statement in the official Java documentation for string. Let us have a look at, open the link in your browser.

<https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/lang/String.html>

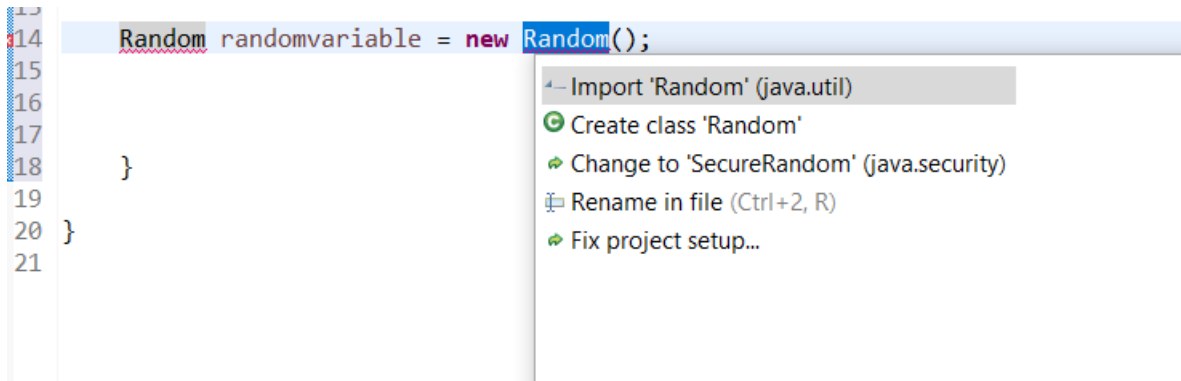
Imagine you want to use a Java library called `Random`. This class is included in the package `java.util`. It provides a set of classes and method to obtain random numbers, Integers, and decimal numbers. Importing and utilizing this library is a simple task in Java, you can follow two ways:

1. Just below the package statement, you can add an import for this class:

```
package oppfundamentals;
```

```
import java.util.Random;
```

2. You can declare a Random variable. As a result, it will come out a compiler error, and the IDE will suggest adding the import, in the case you do not know the import route.



You can employ now this Java class. The usage is like what you made to your own classes. The difference is that you do not know the internal or private code of this class the only you have granted access is to the public properties and methods. The next statement will return a random number in the range of 0 and 20.

```
int randomInt = randomVariable.nextInt(20);
```

Let us look over the import again. This import shows two parts of the import URI:

Package: java.util

Class: Random

```
import java.util.Random;
```

In the event that you would like to import the full package, given that you are doing a extensive use of the package classes, (7 or 8 classes) you can also import the full package, adding after the package URI, dot and Asterisk:

Practice

There is Java class called Date, which belongs to the java.util package, that represent dates.

3. Seek for the Date official documentation
4. Create a Date variable with the current date
5. Print it in the console

5.3 Importing our own classes

As long as your classes are in the same package, they have visibility. It means that they can be used in any other class in the package, while the import sentence is not needed. On the contrary, if you set classes in different packages, they do not have visibility. It surges the necessity of importing the class you are using. In the method Equals section, as a reminder, we set AppEquals.java in a different package. As a result, we needed to add the import to be able to use that class in our code.

```
package appfundamentals;
```

```
import oppfundamentals.SUVElectric;
```

```
public class AppEquals {
```

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

5.4 Wrap classes for primitive types. Type Conversion

Due to Java being an Object-Oriented programming language, Java offers a future to wrap up all primitive types with classes. The purpose is to be able to work in Java programs with classes. These wrap classes are alike to the String class since although they are classes in

some manner they behave as primitive types. Hence, a primitive wrapper class is a wrapper class that encapsulates, hides or wraps data types from the eight primitive data types, so that these can be used to create instantiated objects with methods in another class or in other classes. The primitive wrapper classes are found in the Java API.

The eight basic primitive classes are the following:

- lang. Boolean.
- lang. Character.
- lang. Byte.
- lang. Short.
- lang. Integer.
- lang. Long.
- lang. Float.
- lang. Double.

For example, we have a class to involve and embed the primitive int, call Integer. An object of type Integer contains a single field whose type is int. In addition, this class provides several methods for converting an int to a String and a String to an int, as well as other constants and methods useful when dealing with an int.

Generally speaking, these primitive alike classes, String included in the author's opinion, provide methods to perform type conversion. Also, the constructor could receive one of two parameters from a different type. Remember the overloading feature in Java. It let creating different version of the same method.

Primitive type	Wrapper class	Constructor arguments (old versions of Java) valueOf arguments (from Java 13)
<code>byte</code>	<code>Byte</code>	<code>byte</code> or <code>String</code>
<code>short</code>	<code>Short</code>	<code>short</code> or <code>String</code>
<code>int</code>	<code>Integer</code>	<code>int</code> or <code>String</code>
<code>long</code>	<code>Long</code>	<code>long</code> or <code>String</code>

<code>float</code>	<code>Float</code>	<code>float</code> , <code>double</code> or <code>String</code>
<code>double</code>	<code>Double</code>	<code>double</code> or <code>String</code>
<code>char</code>	<code>Character</code>	<code>char</code>
<code>boolean</code>	<code>Boolean</code>	<code>boolean</code> or <code>String</code>

The Integer class, for instance, is defined similar to the next example:

```
Class Integer {
    private int value;

    public int intValue() {
        return value;
    }
    ... more methods
}
```

You may create objects of this classes by means of the constructor, and also assigning literals, as you did with the String class. It is more common to create this classes with the assign statement. Anyway, provided that you would like to do type conversion, you can use the constructor likewise, in former versions of Java. From Java 13 constructors for these classes are deprecated, in other words, they are recommended not to be used. When Java or other developers tag a method, field or type as deprecated, there are certain constructors, fields, types or methods that they don't want people to use anymore.

Consequently, to carry out type conversion you can use some of the methods provided by this class, such as `valueOf`. i.e `Integer.valueOf("5")`. Primitive wrapper classes are not the same thing as primitive types. Whereas variables, for example, can be declared in Java as data types included `double`, `short`, `int`, etc., the primitive wrapper classes create instantiated objects and methods that inherit but hide the primitive data types, not like variables that are assigned the data type values.

But first we will try to illustrate how to create this wrap objects in modern versions of java.

- The Integer constructor is deprecated, It is not recommended to use.

```
Integer iObj = new Integer("3");
```

```
Float f1Obj = new Float(dObj);
```

- We can assign literals to this class variables, which results in the object automatic object creation.

```
Long lObj = 5L;
Double dObj = 5.9;

Boolean blObj = true;
```

- We can use conversion methods.

```
Integer intVar = Integer.valueOf("6");

Byte bObj = Byte.valueOf("23");
```

- We can get the original primitive value.

```
int iPrim = iObj.intValue();
```

- Similarly, you can assign directly this objects to primitive type variables. As we mentioned before, these classes only have one attribute to store the primitive type. Hence, the compiler allows programmers to work with them as primitive types. You can find the possible argument types to use for `valueOf` in the table above

```
int iPrim = iObj;
long lPrim = lObj;

double dPrim = dObj;
```

Finally, `String` offers a method for `String` conversion, `valueOf`, which can receive all the primitive types as arguments to be converted. This method is overloaded in the `String` class, receiving different parameters. Pay attention to the next chunk of code, so that we are converting from primitive types to `String`.

```
String convPrim = " int to String " + String.valueOf(iPrim) + " " +
    " long to String " + String.valueOf(lPrim) + "." +
    " double to String " + String.valueOf(dPrim) + "." +
    " float to String " + String.valueOf(f1Prim) + "." +
    " boolean to String " + String.valueOf(blPrim) + "." ;

System.out.println("Primitive Types conversion to String" + convPrim);
```

Likewise, `String.valueOf()` can receive all the wrapper types as arguments to be converted, and that results from the structure of Wrappers, given that they store the primitive type in

an attribute. It is illustrated in the next sequence of code:

```
String convObj = " Integer to String " + String.valueOf(iObj) + "." +  
    " Long to String " + String.valueOf(lObj) + "." +  
    " Double to String " + String.valueOf(dObj) + "." +  
    " Float to String " + String.valueOf(fObj) + "." +  
    " Boolean to String " + String.valueOf(bObj) + "." ;
```

```
System.out.println("Primitive Types conversion to String" + convObj);
```

In the following units, we will look through how java deals with primitive types and offer some adapter classes to use them in more complex Java models and classes in the Java API. Neither is to say, that the trend in Java Programming is using wrapper classes instead of primitive types.

6 Index

6.1 Java keywords

Java has a set of keywords that are reserved words that cannot be used as variables, methods, classes, or any other identifiers:

Keyword	Description
abstract	A non-access modifier. Used for classes and methods: An abstract class cannot be used to create objects (to access it, it must be inherited from another class). An abstract method can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from)
assert	For debugging

boolean	A data type that can only store true and false values
break	Breaks out of a loop or a switch block
byte	A data type that can store whole numbers from -128 and 127
case	Marks a block of code in switch statements
catch	Catches exceptions generated by try statements
char	A data type that is used to store a single character
class	Defines a class
continue	Continues to the next iteration of a loop
const	Defines a constant. Not in use - use final instead
default	Specifies the default block of code in a switch statement
do	Used together with while to create a do-while loop
double	A data type that can store whole numbers from 1.7e−308 to 1.7e+308

else	Used in conditional statements
enum	Declares an enumerated (unchangeable) type
exports	Exports a package with a module. New in Java 9
extends	Extends a class (indicates that a class is inherited from another class)
final	A non-access modifier used for classes, attributes, and methods-which makes them non-changeable (impossible to inherit or override)
finally	Used with exceptions, a block of code that will be executed no matter if there is an exception or not
float	A data type that can store whole numbers from 3.4e−038 to 3.4e+038
for	Create a for loop
goto	Not in use, and has no function
if	Makes a conditional statement
implements	Implements an interface

import	Used to import a package, class or interface
instanceof	Checks whether an object is an instance of a specific class or an interface
int	A data type that can store whole numbers from -2147483648 to 2147483647
interface	Used to declare a special type of class that only contains abstract methods
long	A data type that can store whole numbers from -9223372036854775808 to 9223372036854775808
module	Declares a module. New in Java 9
native	Specifies that a method is not implemented in the same Java source file (but in another language)
new	Creates new objects
package	Declares a package
private	An access modifier used for attributes, methods and constructors, making them only accessible within the declared class
protected	An access modifier used for attributes, methods and constructors,

	making them accessible in the same package and subclasses
public	An access modifier used for classes, attributes, methods and constructors, making them accessible by any other class
requires	Specifies required libraries inside a module. New in Java 9
return	Finished the execution of a method, and can be used to return a value from a method
short	A data type that can store whole numbers from -32768 to 32767
static	A non-access modifier used for methods and attributes. Static methods/attributes can be accessed without creating an object of a class
strictfp	Restrict the precision and rounding of floating point calculations
super	Refers to superclass (parent) objects
switch	Selects one of many code blocks to be executed
synchronized	A non-access modifier, which specifies that methods can only be accessed by one thread at a time
this	Refers to the current object in a method or constructor

throw	Creates a custom error
throws	Indicates what exceptions may be thrown by a method
transient	A non-accesss modifier, which specifies that an attribute is not part of an object's persistent state
try	Creates a try...catch statement
var	Declares a variable. New in Java 10
void	Specifies that a method should not have a return value
volatile	Indicates that an attribute is not cached thread-locally, and is always read from the "main memory"
while	Creates a while loop

