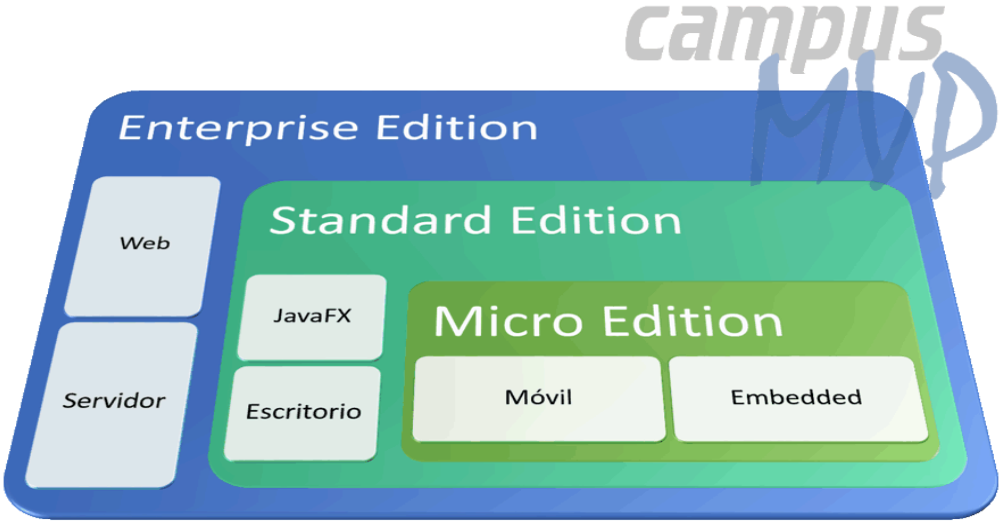
JAVA

Introducción

- Desarrollado por **Sun Microsystems** y adquirido por **ORACLE en 2010**.
    - Su sintaxis deriva en gran medida de C y C++**.** Es un lenguaje de programación de propósito general, concurrente, orientado a objetos, que fue diseñado específicamente para tener tan pocas dependencias de implementación como fuera posible. Su intención es permitir que los desarrolladores de aplicaciones escriban el programa una vez y lo ejecuten en cualquier dispositivo

- **Tres posibles plataformas:**

    - **Java SE:** la conocida como **Standard Edition** es la edición más difundida de la plataforma Java. Incorpora los elementos necesarios para crear **aplicaciones de escritorio** con o sin interfaz gráfica de usuario, acceso al sistema de archivos, comunicación a través de redes, concurrencia y otros servicios básicos.
        - **JavaFX:** originalmente JavaFX era una alternativa a Java SE para el desarrollo de proyectos de tipo RIA (**Rich Internet Applications**), con un núcleo más ligero y fácil de distribuir, capacidad de aceleración 3D aprovechando la GPU, servicios avanzados para producción de gráficos y animaciones, y un mecanismo simplificado para el diseño de interfaces de usuario. JavaFX **forma parte de Java SE desde la versión 7** de dicha edición de la plataforma.
    - **Java EE:** es la **Enterprise Edition** de la plataforma Java, dirigida al desarrollo de soluciones software que se ejecutarán en un **servidor de aplicaciones**. A las capacidades de Java SE, la edición EE agrega servicios para gestionar la persistencia de objetos en bases de datos, hacer posible la invocación remota de métodos, crear aplicaciones con interfaz de usuario web, etc.
    - **Java ME:** esta edición de la plataforma, **Micro Edition**, está enfocada a la creación de programas que se ejecutarán en sistemas con recursos limitados, tales como teléfonos móviles, electrodomésticos y dispositivos de domótica o equipos para entornos empotrados como la Rasperry Pi y similares.



Types of Java Applications

There are mainly 4 types of applications that can be created using Java programming:

1) Standalone Application

Standalone applications are also known as desktop applications or window-based applications. These are traditional software that we need to install on every machine. Examples of standalone application are Media player, antivirus, etc. AWT and Swing are used in Java for creating standalone applications.

2) Web Application

An application that runs on the server side and creates a dynamic page is called a web application. Currently, Servlet, JSP, Struts, Spring, Hibernate, JSF, etc. technologies are used for creating web applications in Java.

3) Enterprise Application

An application that is distributed in nature, such as banking applications, etc. is called an enterprise application. It has advantages like high-level security, load balancing, and clustering. In Java, EJB is used for creating enterprise applications.

4) Mobile Application

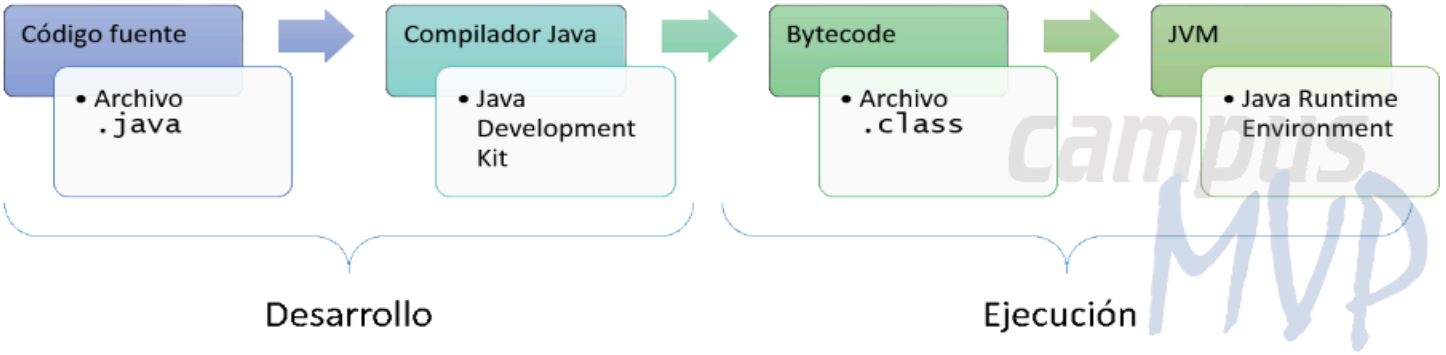
An application which is created for mobile devices is called a mobile application. Currently, Android and Java ME are used for creating mobile applications.

- **Partes**

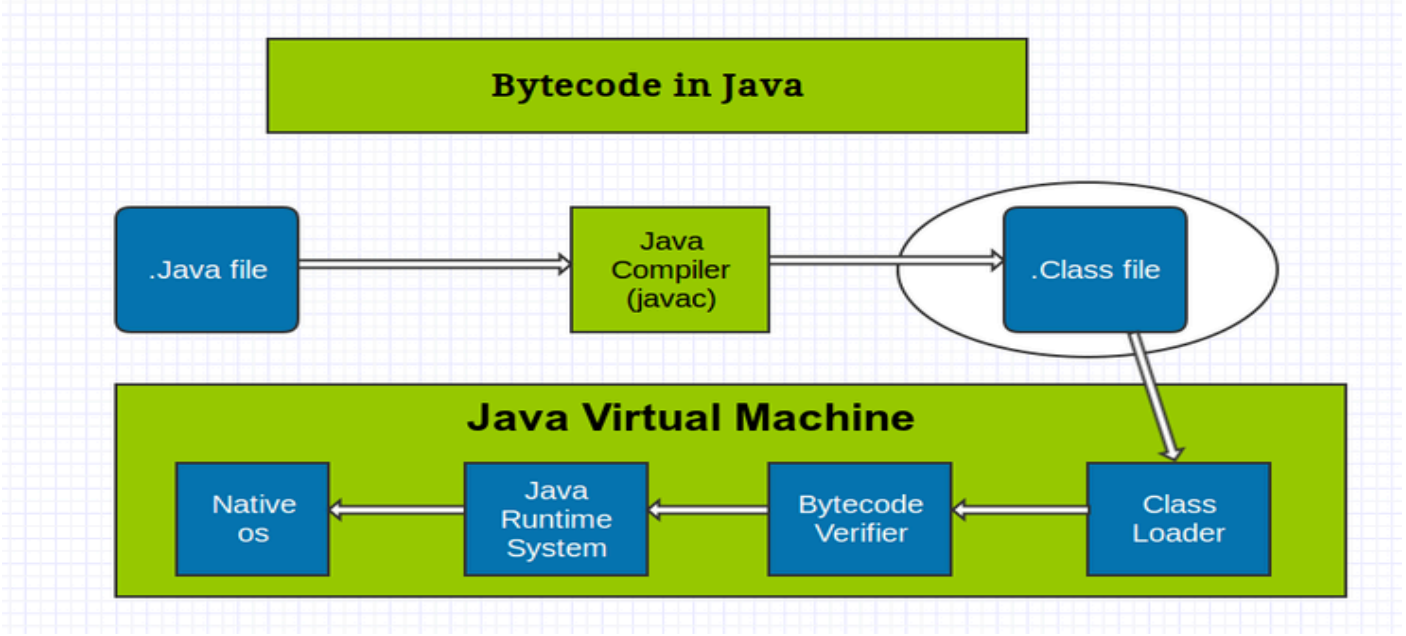
  Dos partes muy diferenciadas dentro de la plataforma Java
  :

  JRE (*Java Runtime Environment*): su objeto es aportar el entorno necesario para ejecutar una aplicación Java. Forman parte del JRE la **máquina virtual Java o JVM** (*Java Virtual Machine*), encargada de ejecutar el *bytecode* Java, así como las bibliotecas que ofrecen los servicios definidos en la plataforma.

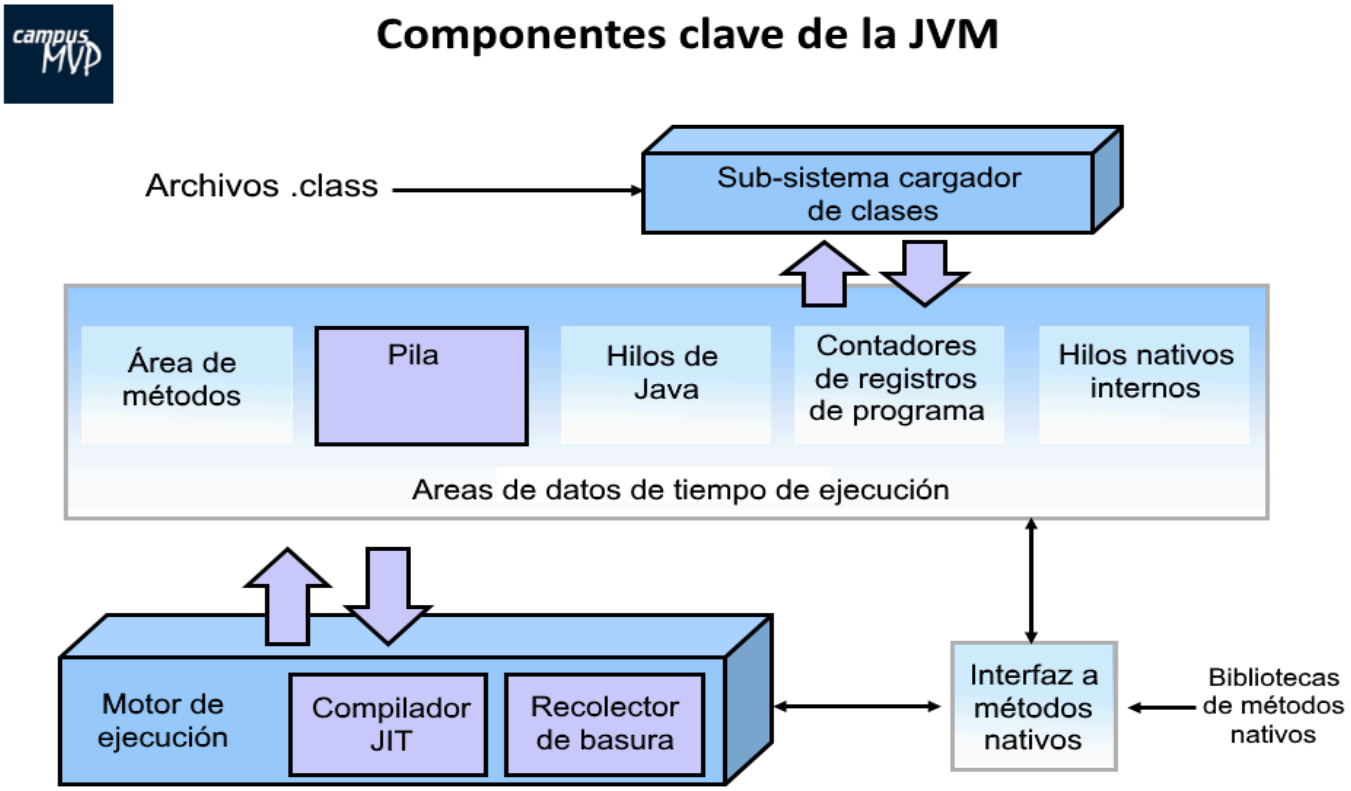
  JDK (*Java Development Kit*): es el paquete de herramientas para llevar a cabo el desarrollo de dicha aplicación. Este JDK es un superconjunto del JRE, al que agrega herramientas como el **compilador Java**. Este toma el código fuente Java y genera como resultado *bytecode*, un formato de código objeto independiente del sistema operativo y el hardware.



- **Máquina Virtual de Java JVM (Java Virtual Machine)**
  - Cuando compilas una aplicación escrita en lenguaje Java, en realidad éste no se compila a lenguaje máquina directamente entendible por el sistema operativo, sino a un lenguaje intermedio denominado **Byte Code**. Lo mismo ocurre con las aplicaciones .NET que se compilan también a un lenguaje intermedio llamado MSIL (MicroSoft Intermediate Language).
  - Entre el Byte Code (o el MSIL en el caso de .NET) y el sistema operativo se coloca un componente especial llamado **Máquina virtual** que es el que realmente va a ejecutar el código. En el caso de Java, La Java Virtual Machine o JVM toma el código Byte Code resultante de compilar tu aplicación Java y lo compila a su vez a código nativo de la plataforma en la que se está ejecutando. La ventaja principal de este esquema es que es muy fácil crear un programa en Java y que luego éste se pueda ejecutar en cualquier sistema operativo para el cual exista una implementación de la JVM.
  - **Write Once, Run Anywhere**



- **El Compilador Just In-Time o JIT o Compilador en tiempo real :**
  - La ejecución del Byte Code es lenta comparada con la del código nativo en cada plataforma. La JVM primero tiene que cargar el código y traducirlo a instrucciones nativas para poder ejecutarlo, haciendo otras muchas cosas por el medio, como comprobar los tipos, recoger basura, etc… Una manera de optimizarlo y conseguir un alto rendimiento es hacer una compilación inmediata, lo antes posible, al código nativo de cada plataforma, cacheando además los resultados para reutilizarlos. Esto es básicamente lo que hace este componente, cuyo nombre viene de esta función: Just In-Time Compiler o compilador en tiempo real. OJO, esto no quiere decir que compile todo el código a código nativo (para eso tendríamos un compilador para cada plataforma, como ocurre con C++, por ejemplo). Lo que hace la JVM es decidir en cada momento qué partes del código se deben compilar con el JIT y cuáles almacenar, cuándo ejecutarlas, etc...Dado que esa compilación también lleva tiempo, el compilador JIT suele compilar a código nativo el código que se use con mayor frecuencia, pudiendo notar una pequeña demora en la ejecución la primera vez que se compila.



  -

INSTALACIÓN

1. Java es gratuita , simplemente dirígete a ORACLE para bajar el JAVA **JRE** (J.Runtime Enviroment)
   a. **https://www.java.com/es/download/**
   b. **https://www.java.com/es/download/help/**
2. Para comprobar que java está instalado teclear desde la línea de comandos:
   java -version
3. A veces es necesario añadir a la variable PATH la ruta de instalación, que termina en \bin:
   a. PATH = C:\Program Files\Java\jdk-11.0.1**\bin**

EMPEZANDO A PROGRAMAR

1. Para poder programar con JAVA necesitas el JAVA **JDK (J.Development Kit)**
   a. **https://www.oracle.com/technetwork/es/java/javase/downloads/index.html**
   b. Documentación de Oracle Java Documentation - Get Started (oracle.com)
   c. Free learning Oracle Oracle Learning Explorer: Learn Oracle for Free | Oracle University.

2. Para comenzar a programar en Java nos basta con un editor de **texto y JDK** instalado

3. **Los IDEs** (*Integrated Development Environment*) recomendados para programar JAVA son:
   a. **NeatBeans (Oracle)**
   b. **Visual Studio Code, Eclipse, IntelliJ IDEA**
   c. Editores más ligeros: **Geany , BlueJ**
   d. Desde un editor se puede compilar y luego ejecutar.

4. Todo programa Java debe contener al menos **una clase**.

5. El archivo de código fuente tiene extensión **.java** **MyClass.java** y **debe de coincidir** con el nombre de la clase principal o top level.

```
public class MyClass {
  public static void main(String[] args) {
    System.out.println("Hello World");
  }
}
```

6. **Para compilar** el archivo de código fuente y generar el **ByteCode**:
   a. Desde la línea de comando **javac MyClass.java**
   b. El compilador genera un archivo **MyClass.class**
   c. Para ejecutarlo hay que teclear: **java MyClass**
   d. También se puede hacer desde un **IDE o Editor** : **compilar y luego ejecutar.**

7. **Para compilar desde GEANY**
   a. Primero pulsar icono **Compilar , Geany generará el archivo .class junto al archivo .java que estamos editando.**
   b. Luego pulsar **Ejecutar** para probar el archivo **.class** generado.

8. **Sintaxis básica :**
   a. Se usa UTF-8.
   b. Google JAVA style guide https://google.github.io/styleguide/javaguide.html
   c. El nombre de la clase debe empezar por **mayúscula MyClass().**
   d. JAVA es **casesensitive**
   e. Todo programa debe tener un método **main()**
   f. Los métodos empiezan por minúsculas **myMethodName()**
   g. **Identificadores** para Clases , Variables y Métodos:
      Pueden contener a-zA-Z$_0-9 y deben empezar por a-zA-Z$_

9. **Comentarios**
    a. // una solo línea
    b. /* varias líneas */.
    c. /** Comentarios para Java DOC */

10. **Variables**
    a. Local Variables
    b. Class Variables (Static Variables)
    c. Instance Variables (Non-static Variables)

11. **Modificadores**
    a. Access Modifiers − **default, public , protected, private**
    b. Non-access Modifiers − **final, abstract, strictfp**

12. **KeyWords**

abstract: Java abstract keyword is used to declare an abstract class. An abstract class can provide the implementation of the interface. It can have abstract and non-abstract methods.

boolean: Java boolean keyword is used to declare a variable as a boolean type. It can hold True and False values only.

break: Java break keyword is used to break the loop or switch statement. It breaks the current flow of the program at specified conditions.

byte: Java byte keyword is used to declare a variable that can hold 8-bit data values.

case: Java case keyword is used with the switch statements to mark blocks of text.

catch: Java catch keyword is used to catch the exceptions generated by try statements. It must be used after the try block only.

char: Java char keyword is used to declare a variable that can hold unsigned 16-bit Unicode characters

class: Java class keyword is used to declare a class.

continue: Java continue keyword is used to continue the loop. It continues the current flow of the program and skips the remaining code at the specified condition.

default: Java default keyword is used to specify the default block of code in a switch statement.

do: Java do keyword is used in the control statement to declare a loop. It can iterate a part of the program several times.

double: Java double keyword is used to declare a variable that can hold 64-bit floating-point number.

else: Java else keyword is used to indicate the alternative branches in an if statement.

enum: Java enum keyword is used to define a fixed set of constants. Enum constructors are always private or default.

extends: Java extends keyword is used to indicate that a class is derived from another class or interface.

final: Java final keyword is used to indicate that a variable holds a constant value. It is used with a variable. It is used to restrict the user from updating the value of the variable.

finally: Java finally keyword indicates a block of code in a try-catch structure. This block is always executed whether an exception is handled or not.

float: Java float keyword is used to declare a variable that can hold a 32-bit floating-point number.

for: Java for keyword is used to start a for loop. It is used to execute a set of instructions/functions repeatedly when some condition becomes true. If the number of iteration is fixed, it is recommended to use for loop.

if: Java if keyword tests the condition. It executes the if block if the condition is true.

implements: Java implements keyword is used to implement an interface.

import: Java import keyword makes classes and interfaces available and accessible to the current source code.

instanceof: Java instanceof keyword is used to test whether the object is an instance of the specified class or implements an interface.

int: Java int keyword is used to declare a variable that can hold a 32-bit signed integer.

interface: Java interface keyword is used to declare an interface. It can have only abstract methods.

long: Java long keyword is used to declare a variable that can hold a 64-bit integer.

native: Java native keyword is used to specify that a method is implemented in native code using JNI (Java Native Interface).

new: Java new keyword is used to create new objects.

null: Java null keyword is used to indicate that a reference does not refer to anything. It removes the garbage value.

package: Java package keyword is used to declare a Java package that includes the classes.

private: Java private keyword is an access modifier. It is used to indicate that a method or variable may be accessed only in the class in which it is declared.

protected: Java protected keyword is an access modifier. It can be accessible within the package and outside the package but through inheritance only. It can't be applied with the class.

public: Java public keyword is an access modifier. It is used to indicate that an item is accessible anywhere. It has the widest scope among all other modifiers.

return: Java return keyword is used to return from a method when its execution is complete.

short: Java short keyword is used to declare a variable that can hold a 16-bit integer.

static: Java static keyword is used to indicate that a variable or method is a class method. The static keyword in Java is mainly used for memory management.

strictfp: Java strictfp is used to restrict the floating-point calculations to ensure portability.

super: Java super keyword is a reference variable that is used to refer to parent class objects. It can be used to invoke the immediate parent class method.

switch: The Java switch keyword contains a switch statement that executes code based on test value. The switch statement tests the equality of a variable against multiple values.

synchronized: Java synchronized keyword is used to specify the critical sections or methods in multithreaded code.

this: Java this keyword can be used to refer the current object in a method or constructor.

throw: The Java throw keyword is used to explicitly throw an exception. The throw keyword is mainly used to throw custom exceptions. It is followed by an instance.

throws: The Java throws keyword is used to declare an exception. Checked exceptions can be propagated with throws.

transient: Java transient keyword is used in serialization. If you define any data member as transient, it will not be serialized.

try: Java try keyword is used to start a block of code that will be tested for exceptions. The try block must be followed by either catch or finally block.

void: Java void keyword is used to specify that a method does not have a return value.

volatile: Java volatile keyword is used to indicate that a variable may change asynchronously.

while: Java while keyword is used to start a while loop. This loop iterates a part of the program several times. If the number of iteration is not fixed, it is recommended to use the while loop.

13. Escribiendo en la consola:

a. **System.out.print()**
   i. System.out.**print**(“Imprime texto sin salto de línea”)

b. **System.out.println()**
   i. System.out.**println**(“Imprime texto con salto de línea al final”)

c. **System.out.printf()** (salida formateada similar al lenguaje C printf())
   i. %c un carácter
   ii. %s String
   iii. %d número entero
   iv. %f número con decimales FLOAT
   v.
```
System.out.printf("%-10s %8.2f %6d\n", "manzanas", 4.5, 10);
System.out.printf("%-10s %8.2f %6d\n", "peras", 2.75, 120);
System.out.printf("%-10s %8.2f %6d\n", "aguacates", 10.0, 6);
```
   vi.
```
manzanas     4,50    10
peras        2,75   120
aguacates   10,00     6
```

14. Caracteres de escape

a. \n salto de línea \t tabulador \” \’ \\

15. Caracteres especiales

a. http://www.unicode.org/charts/ **Unicode** : de \u0000 a \uFFFF

16. Coloreado de texto por la salida de la consola

a. System.*out*.println("**\033**[31mEste texto es Rojo");
b. System.*out*.println("**\033**[32mEste texto es Verde");
c. //public static final String BLACK="\033[30m"; RED="\033[31m"; GREEN="\033[32m"; YELLOW ="\033[33m"; BLUE = "\033[34m"; PURPLE = "\033[35m"; CYAN = "\033[36m"; WHITE = "\033[37m";
d. El color antes de la cadena System.*out*.println(**color** + txt);
e. System.*out*.println(**RED** +”Hola soy rojo”);
f. System.*out*.println(**RED** +”Hola soy rojo” + **BLUE** + ‘Soy azul’);

17. Nombres de variables

a. Las variables se crean con estilo lowerCamelCase.
b. Puede contener números y guiones bajos , pero no empezar por número.
c. https://google.github.io/styleguide/javaguide.html

18. Tipos de datos:

a. boolean(1) true or false
b. byte(1) Stores whole numbers from -128 to 127
c. short(2) Stores whole numbers from -32,768 to 32,767
d. int(4) Stores whole numbers from -2,147,483,648 to 2,147,483,647
e. long(8) -9,223,372,036,854,775,808 to 9,223,372,036,854,775,808
f. float(4) Fractional numbers from 3.4e−038 to 3.4e+038
g. double(8) 1.7e−308 to 1.7e+038
h. char(2) Stores a single character/letter or ASCII values

i. String
j. Arrays
k. Classes

19. Ejemplos

a. `int myNum = 5;`
b. `char myLetter = 'D';`
c. `char myVar1 = 65;`
d. `boolean myBool = true;`
e. `String myText = "Hello";`
f. `int x = 5, y = 6, z = 50;`
g. `int x, y, z;`
h. `x = y = z = 50;`
i. `long myNum = 15000000000L; //OJO  L al final`
j. `float myNum = 5.75f; //OJO  f al final`
k. `double myNum = 19.99d; //OJO  d al final`
l.
m. `final int myNum = 15;  (final=Valor constante no modificable)`

20. Java Type Casting

- **Widening Casting** (automatically) - converting a smaller type to a larger type size
  `byte` -> `short` -> `char` -> `int` -> `long` -> `float` -> `double`
- **Narrowing Casting** (manually) - converting a larger type to a smaller size type
  `double` -> `float` -> `long` -> `int` -> `char` -> `short` -> `byte`

Widening casting is done automatically

```
public class Main {
  public static void main(String[] args) {
    int myInt = 9;
    double myDouble = myInt; // Automatic casting: int to double

    System.out.println(myInt);      // Outputs 9
    System.out.println(myDouble);   // Outputs 9.0
  }
}
```

Narrowing casting must be done manually

```
public class Main {
  public static void main(String[] args) {
    double myDouble = 9.78d;
    int myInt = (int) myDouble; // Manual casting: double to int

    System.out.println(myDouble);   // Outputs 9.78
    System.out.println(myInt);      // Outputs 9
  }
```

Otro Ejemplo:

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

    int x = 2;
    int y = 9;
    double division;

    division = (double)y / (double)x;
    //division = y / x; // Comenta esta línea y
                // observa la diferencia.

    System.out.println("El resultado de la división es " + division);
  }
}
```

21. OPERADORES

Arithmetic Operators

Arithmetic operators are used to perform common mathematical operations.

Operator	Name	Description	Example
+	Addition	Adds together two values	x + y
-	Subtraction	Subtracts one value from another	x - y
*	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

A list of all assignment operators:

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Java Comparison Operators

Comparison operators are used to compare two values:

Operator	Name	Example
==	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

Java Logical Operators

Logical operators are used 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	x < 5 || x < 4
!	Logical not	Reverse the result, returns false if the result is true	!(x < 5 && x < 10)

22. Strings [Java String Reference (w3schools.com)]

```
String txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
System.out.println("The length of the txt string is: " + txt.length());
String txt = "Hello World";
System.out.println(txt.toUpperCase());   // Outputs "HELLO WORLD"
System.out.println(txt.toLowerCase());   // Outputs "hello world"
String txt = "Please locate where 'locate' occurs!";
System.out.println(txt.indexOf("locate")); // Outputs 7
```

charAt()	Returns the character at the specified index (position)	char
codePointAt()	Returns the Unicode of the character at the specified index	int
codePointBefore()	Returns the Unicode of the character before the specified index	int
codePointCount()	Returns the number of Unicode values found in a string.	int
compareTo()	Compares two strings lexicographically	int
compareToIgnoreCase()	Compares two strings lexicographically, ignoring case differences	int
concat()	Appends a string to the end of another string	String
contains()	Checks whether a string contains a sequence of characters	boolean
contentEquals()	Checks whether a string contains the exact same sequence of characters of the specified CharSequence or StringBuffer	boolean
copyValueOf()	Returns a String that represents the characters of the character array	String
endsWith()	Checks whether a string ends with the specified character(s)	boolean
equals()	Compares two strings. Returns true if the strings are equal, and false if not	boolean
equalsIgnoreCase()	Compares two strings, ignoring case considerations	boolean
format()	Returns a formatted string using the specified locale, format string, and arguments	String
getBytes()	Encodes this String into a sequence of bytes using the named charset, storing the result into a new byte array	byte[]
getChars()	Copies characters from a string to an array of chars	void
hashCode()	Returns the hash code of a string	int
indexOf()	Returns the position of the first found occurrence of specified characters in a string	int
intern()	Returns the canonical representation for the string object	String
isEmpty()	Checks whether a string is empty or not	boolean
lastIndexOf()	Returns the position of the last found occurrence of specified characters in a string	int
length()	Returns the length of a specified string	int
matches()	Searches a string for a match against a regular expression, and returns the matches	boolean
offsetByCodePoints()	Returns the index within this String that is offset from the given index by codePointOffset code points	int
regionMatches()	Tests if two string regions are equal	boolean
replace()	Searches a string for a specified value, and returns a new string where the specified values are replaced	String
replaceFirst()	Replaces the first occurrence of a substring that matches the given regular expression with the given replacement	String
replaceAll()	Replaces each substring of this string that matches the given regular expression with the given replacement	String
split()	Splits a string into an array of substrings	String[]
startsWith()	Checks whether a string starts with specified characters	boolean
subSequence()	Returns a new character sequence that is a subsequence of this sequence	CharSequence
substring()	Returns a new string which is the substring of a specified string	String
toCharArray()	Converts this string to a new character array	char[]
toLowerCase()	Converts a string to lower case letters	String
toString()	Returns the value of a String object	String
toUpperCase()	Converts a string to upper case letters	String
trim()	Removes whitespace from both ends of a string	String
valueOf()	Returns the string representation of the specified value	String

23. Math

```
System.out.println(Math.abs(4.7));
```

abs(x)	Returns the absolute value of x	double|float|int|long
acos(x)	Returns the arccosine of x, in radians	double
asin(x)	Returns the arcsine of x, in radians	double
atan(x)	Returns the arctangent of x as a numeric value between -PI/2 and PI/2 radians	double
atan2(y,x)	Returns the angle theta from the conversion of rectangular coordinates (x, y) to polar coordinates (r, theta).	double
cbrt(x)	Returns the cube root of x	double
ceil(x)	Returns the value of x rounded up to its nearest integer	double
copySign(x, y)	Returns the first floating point x with the sign of the second floating point y	double
cos(x)	Returns the cosine of x (x is in radians)	double
cosh(x)	Returns the hyperbolic cosine of a double value	double
exp(x)	Returns the value of Ex	double
expm1(x)	Returns ex -1	double
floor(x)	Returns the value of x rounded down to its nearest integer	double
getExponent(x)	Returns the unbiased exponent used in x	int
hypot(x, y)	Returns sqrt(x2 +y2) without intermediate overflow or underflow	double
IEEEremainder(x, y)	Computes the remainder operation on x and y as prescribed by the IEEE 754 standard	double
log(x)	Returns the natural logarithm (base E) of x	double
log10(x)	Returns the base 10 logarithm of x	double
log1p(x)	Returns the natural logarithm (base E) of the sum of x and 1	double
max(x, y)	Returns the number with the highest value	double|float|int|long
min(x, y)	Returns the number with the lowest value	double|float|int|long
nextAfter(x, y)	Returns the floating point number adjacent to x in the direction of y	double|float
nextUp(x)	Returns the floating point value adjacent to x in the direction of positive infinity	double|float
pow(x, y)	Returns the value of x to the power of y	double
random()	Returns a random number between 0 and 1	double
round(x)	Returns the value of x rounded to its nearest integer	int
rint(x)	Returns the double value that is closest to x and equal to a mathematical integer	double
signum(x)	Returns the sign of x	double
sin(x)	Returns the sine of x (x is in radians)	double
sinh(x)	Returns the hyperbolic sine of a double value	double
sqrt(x)	Returns the square root of x	double
tan(x)	Returns the tangent of an angle	double
tanh(x)	Returns the hyperbolic tangent of a double value	double
toDegrees(x)	Converts an angle measured in radians to an approx. equivalent angle measured in degrees	double
toRadians(x)	Converts an angle measured in degrees to an approx. angle measured in radians	double
ulp(x)	Returns the size of the unit of least precision (ulp) of x	double|float

24. If , Switch , While , For

```
if (condition1) {
  // block of code to be executed if condition1 is true
} else if (condition2) {
  // block of code to be executed if the condition1 is false and condition2 is true
} else {
  // block of code to be executed if the condition1 is false and condition2 is false
}
```

operador ?

```
variable = (condition) ? expressionTrue : expressionFalse;
String result = (time < 18) ? "Good day." : "Good evening.";
```

```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

```
while (condition) {
  // code block to be executed
}
```

```
do {
  // code block to be executed
}
while (condition);
```

```
for (statement 1; statement 2; statement 3) {
  // code block to be executed
}
```

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
for (String i : cars) {  // FOR-EACH in Array
  System.out.println(i);
}
```

25. Arrays

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
int[] myNum = {10, 20, 30, 40};
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
cars[0] = "Opel";
System.out.println(cars[0]);
System.out.println(cars.length);
int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };
int x = myNumbers[1][2];
int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };
    for (int i = 0; i < myNumbers.length; ++i) {
      for(int j = 0; j < myNumbers[i].length; ++j) {
        System.out.println(myNumbers[i][j]);
      }
   }
```

26. Java Methods

```
public class MyMain {
  static void myMethod(String fname, int age) {
    System.out.println(fname + " is " + age);
  }

  public static void main(String[] args) {
    myMethod("Liam", 5);
    myMethod("Jenny", 8);
    myMethod("Anja", 31);
  }
}
```

- `myMethod()` is the name of the method
- `static` means that the method belongs to the MyMain class and not an object of the MyMain class.
- `void` means that this method does not have a return value.

Return

```
public class MyMain {
  static int myMethod(int x, int y) {
    return x + y;
  }

  public static void main(String[] args) {
    int z = myMethod(5, 3);
    System.out.println(z);
  }
}

// Outputs 8 (5 + 3)
```

Overloading (Polimorfismo overload)

With method overloading, multiple methods can have the same name with **different parameters (different data types and /or number of parameters)**:

```
public class MyMain {
    static int plusMethod(int x, int y) {
      return x + y;
    }

    static double plusMethod(double x, double y) {
      return x + y;
    }

    public static void main(String[] args) {
      int myNum1 = plusMethod(8, 5);
      double myNum2 = plusMethod(4.3, 6.26);
      System.out.println("int: " + myNum1);
      System.out.println("double: " + myNum2);
    }
}
```

Scope

```
public class MyMain {
  public static void main(String[] args) {
    // Code here CANNOT use x
    { // This is a block
      // Code here CANNOT use x
      int x = 100;
      // Code here CAN use x
      System.out.println(x);
    } // The block ends here
  // Code here CANNOT use x
  }
}
```

Java Clases

```
public class MyMain {
  String fname = "John"; // atributos de la clase
  String lname = "Doe";
  int age = 24;

  public static void main(String[] args) {
    Main myObj = new MyMain();
    System.out.println("Name: " + myObj.fname + " " + myObj.lname);
    System.out.println("Age: " + myObj.age);
  }

}
```

`static` y `public`

An example to demonstrate the differences between `static` and `public` methods:

```
public class Main {
  // Static method
  static void myStaticMethod() {
    System.out.println("Static methods can be called without creating
objects");
  }

  // Public method
  public void myPublicMethod() {
    System.out.println("Public methods must be called by creating objects");
  }

  // Main method
  public static void main(String[] args) {
    myStaticMethod(); // Call the static method
    // myPublicMethod(); This would compile an error

    Main myObj = new Main(); // Create an object of Main
    myObj.myPublicMethod(); // Call the public method on the object
  }

}
```

Using Multiple Classes

Like we specified in the Classes chapter, it is a good practice to create an object of a class and access it in another class.

Remember that the name of the java file should match the class name. In this example, we have created two files in the same directory:

- Main.java
- Second.java

Main.java

```
public class Main {
  public void fullThrottle() {
    System.out.println("The car is going as fast as it can!");
  }

  public void speed(int maxSpeed) {
    System.out.println("Max speed is: " + maxSpeed);
  }
}
```

Second.java

```
class Second {
  public static void main(String[] args) {
    Main myCar = new Main();      // Create a myCar object
    myCar.fullThrottle();       // Call the fullThrottle() method
    myCar.speed(200);           // Call the speed() method
  }
}
```

When both files have been compiled:

```
C:\Users\Your Name>javac Main.java

C:\Users\Your Name>javac Second.java
```

Run the Second.java file:

```
C:\Users\Your Name>java Second
```

And the output will be:

```
The car is going as fast as it can!

Max speed is: 200
```

Constructor method

```
public class Main {
  int modelYear;
  String modelName;
                                    // no es obligatorio tener un constructor
  public Main(int year, String name) { //constructor parametrizado
    modelYear = year;
    modelName = name;
  }

  public static void main(String[] args) {
    Main myCar = new Main(1969, "Mustang");
    System.out.println(myCar.modelYear + " " + myCar.modelName);
  }

}
```

Java Modifiers

By now, you are quite familiar with the `public` keyword that appears in almost all of our examples:

```
public class Main
```

The **`public`** keyword is an access modifier, meaning that it is used to set the access level for classes, attributes, methods and constructors.

We divide modifiers into two groups:

- **Access Modifiers** - controls the access level
- **Non-Access Modifiers** - do not control access level, but provides other functionality

---

Access Modifiers

For classes, you can use either `public` or *default*:

Modifier	Description	Try it
`public`	The class is accessible by any other class	Try it »
default	The class is only accessible by classes in the same package. This is used when you don't specify a modifier. You will learn more about packages in the Packages chapter	Try it »

For attributes, methods and constructors, you can use the one of the following:

Modifier	Description	Try it
`public`	The code is accessible for all classes	Try it »
`private`	The code is only accessible within the declared class	Try it »
default	The code is only accessible in the same package. This is used when you don't specify a modifier. You will learn more about packages in the Packages chapter	Try it »
`protected`	The code is accessible in the same package and subclasses. You will learn more about subclasses and superclasses in the Inheritance chapter	Try it »

Non-Access Modifiers

For classes, you can use either `final` or `abstract`:

Modifier	Description	Try it
`final`	The class cannot be inherited by other classes (You will learn more about inheritance in the Inheritance chapter)	Try it »
`abstract`	The class cannot be used to create objects (To access an abstract class, it must be inherited from another class. You will learn more about inheritance and abstraction in the Inheritance and Abstraction chapters)	Try it »

For attributes and methods, you can use the one of the following:

Modifier	Description
`final`	**Attributes and methods** cannot be overridden/modified
`static`	**Attributes and methods** belongs to the class, rather than an object
`abstract`	Can only be used in an abstract class, and can only be used on methods. The method does not have a body, for example abstract void run();. The body is provided by the subclass (inherited from). You will learn more about inheritance and abstraction in the Inheritance and Abstraction chapters
`transient`	**Attributes and methods** are skipped when serializing the object containing them
`synchronized`	Methods can only be accessed by one thread at a time
`volatile`	The value of an attribute is not cached thread-locally, and is always read from the "main memory"

Naming Conventions of the Different Identifiers

The following table shows the popular conventions used for the different identifiers.

Identifiers Type	Naming Rules	Examples

Class	It should start with the uppercase letter.
	It should be a noun such as Color, Button, System, Thread, etc.
	Use appropriate words, instead of acronyms.
		public class **Employee**
		{
		//code snippet
		}

Interface	It should start with the uppercase letter.
	It should be an adjective such as Runnable, Remote, ActionListener.
	Use appropriate words, instead of acronyms.
		interface **Printable**
		{
		//code snippet
		}

Method	It should start with lowercase letter.
	It should be a verb such as main(), print(), println().
	If the name contains multiple words, start it with a lowercase letter followed by an uppercase letter such as actionPerformed().
		class Employee
		{
		// method
		void **draw()**
		{
		//code snippet
		}
		}

Variable	It should start with a lowercase letter such as id, name.
	It should not start with the special characters like & (ampersand), $ (dollar), _ (underscore).
	If the name contains multiple words, start it with the lowercase letter followed by an uppercase letter such as firstName, lastName.
	Avoid using one-character variables such as x, y, z.
		class Employee
		{
		// variable
		int **id**;
		//code snippet
		}

Package	It should be a lowercase letter such as java, lang.
	If the name contains multiple words, it should be separated by dots (.) such as java.util, java.lang.
		//package
		package **com.javatpoint;**
		class Employee
		{
		//code snippet
		}

Constant	It should be in uppercase letters such as RED, YELLOW.
	If the name contains multiple words, it should be separated by an underscore(_) such as MAX_PRIORITY.
	It may contain digits but not as the first letter.
		class Employee
		{
		//constant
		static final int **MIN_AGE** = 18;
		//code snippet
		}

CamelCase in Java naming conventions

Java follows camel-case syntax for naming the class, interface, method, and variable.

If the name is combined with two words, the second word will start with uppercase letter always such as actionPerformed(), firstName, ActionEvent, ActionListener, etc.

Java Encapsulation

The meaning of Encapsulation, is to make sure that "sensitive" data is hidden from users. To achieve this, you must:

- declare class variables/attributes as **`private`**
- provide public **get and set** methods to access and update the value of a `private` variable

---

Get and Set

You learned from the previous chapter that `private` variables can only be accessed within the same class (an outside class has no access to it). However, it is possible to access them if we provide public get and set methods. The `get` method returns the variable value, and the `set` method sets the value. Syntax for both is that they start with either `get` or `set`, followed by the name of the variable, with the first letter in upper case:

```
public class Person {

  private String name; // private = restricted access

  // Getter
  public String getName() {
    return name;
  }

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

Java Packages & API

A package in Java is used to group related classes. Think of it as a folder in a file directory. We use packages to avoid name conflicts, and to write a better maintainable code. Packages are divided into two categories:

- Built-in Packages (packages from the Java API)
- User-defined Packages (create your own packages)

Built-in Packages

The Java API is a library of prewritten classes, that are free to use, 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/javase/8/docs/api/.**

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:

```
import package.name.Class;   // Import a single class
import package.name.*;   // Import the whole package
```

Import a Class

If you find a class you want to use, for example, the `Scanner` class, which is used to get user input, write the following code:

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

In the example above, `java.util` is a package, while `Scanner` is a class of the `java.util` package. To use the `Scanner` class, create an object of the class and use any of the available methods found in the `Scanner` class documentation. In our example, we will use the `nextLine()` method, which is used to read a complete line:

Using the `Scanner` class to get user input:

```
import java.util.Scanner;

class MyClass {
  public static void main(String[] args) {
    Scanner myObj = new Scanner(System.in);
    System.out.println("Enter username");

    String userName = myObj.nextLine();
    System.out.println("Username is: " + userName);
  }
}
```

Import a Package

There are many packages to choose from. In the previous example, we used the `Scanner` class from the `java.util` package. This package also contains date and time facilities, random-number generator and other utility classes.

To import a whole package, end the sentence with an asterisk sign (`*`). The following example will import ALL the classes in the `java.util` package:

```
import java.util.*;
```

User-defined Packages

To create your own package, you need to understand that Java uses a file system directory to store them. Just like folders on your computer:

```
└── root
    └── mypack
        └── MyPackageClass.java
```

To create a package, use the **`package`** keyword:

MyPackageClass.java

```
package mypack;
class MyPackageClass {
  public static void main(String[] args) {
    System.out.println("This is my package!");
  }
}
```

Save the file as MyPackageClass.java, and compile it:

```
C:\Users\Your Name>javac MyPackageClass.java
```

Then compile the package:

```
C:\Users\Your Name>javac -d . MyPackageClass.java
```

This forces the compiler to create the "mypack" package. The `-d` keyword specifies the destination for where to save the class file. You can use any directory name, like c:/user (windows), or, if you want to keep the package within the same directory, you can use the dot sign ".", like in the example above. Note: The package name should be written in lower case to avoid conflict with class names. When we compiled the package in the example above, a new folder was created, called "mypack". To run the MyPackageClass.java file, write the following:

```
C:\Users\Your Name>java mypack.MyPackageClass
```

The output will be:

```
This is my package!
```

Java Inheritance (Subclass and Superclass)

In Java, it is possible to inherit attributes and methods from one class to another. We group the "inheritance concept" into two categories:

- **subclass** (child) - the class that inherits from another class
- **superclass** (parent) - the class being inherited from

To inherit from a class, use the `extends` keyword.

In the example below, the `Car` class (subclass) inherits the attributes and methods from the `Vehicle` class (superclass):

```
class Vehicle {

  protected String brand = "Ford";         // Vehicle attribute

  public void honk() {                     // Vehicle method
    System.out.println("Tuut, tuut!");
  }
}

class Car extends Vehicle {
  private String modelName = "Mustang";    // Car attribute
  public static void main(String[] args) {

    // Create a myCar object
    Car myCar = new Car();

    // Call the honk() method (from the Vehicle class) on the myCar object
    myCar.honk();

     // Display the value of the brand attribute (from the Vehicle class) and
the value of the modelName from the Car class
    System.out.println(myCar.brand + " " + myCar.modelName);
  }
}
```

Did you notice the `protected` modifier in Vehicle? We set the brand attribute in Vehicle to a `protected` access modifier. If it was set to `private`, the Car class would not be able to access it.

The final Keyword

If you don't want other classes to inherit from a class, use the `final` keyword:

```
final class Vehicle {
  ...}
```

Java Polymorphism (Overriding)

Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance. Like we specified in the previous chapter; Inheritance lets us inherit attributes and methods from another class. Polymorphism uses those methods to perform different tasks. This allows us to perform a single action in different ways. For example, think of a superclass called `Animal` that has a method called `animalSound()`. Subclasses of Animals could be Pigs, Cats, Dogs, Birds - And they also have their own implementation of an animal sound (the pig oinks, and the cat meows, etc.):

```
class Animal {

  public void animalSound() {
    System.out.println("The animal makes a sound");
  }
}

class Pig extends Animal {
  public void animalSound() {
    System.out.println("The pig says: wee wee");
  }
}

class Dog extends Animal {
  public void animalSound() {
    System.out.println("The dog says: bow wow");
  }
}
```

Remember from the Inheritance chapter that we use the `extends` keyword to inherit from a class. Now we can create `Pig` and `Dog` objects and call the `animalSound()` method on both of them:

```
class Animal {
  public void animalSound() {
    System.out.println("The animal makes a sound");
  }
}
class Pig extends Animal {
  public void animalSound() {
    System.out.println("The pig says: wee wee");
  }
}
class Dog extends Animal {
  public void animalSound() {
    System.out.println("The dog says: bow wow");
  }
}
class Main {
  public static void main(String[] args) {
    Animal myAnimal = new Animal();  // Create a Animal object
    Animal myPig = new Pig();  // Create a Pig object
    Animal myDog = new Dog();  // Create a Dog object
    myAnimal.animalSound();
    myPig.animalSound();
    myDog.animalSound();
  }
```

1) **super** is used to refer immediate parent class instance variable.

We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

```
1. class Animal{
2. String color="white";
3. }
4. class Dog extends Animal{
5. String color="black";
6. void printColor(){
7. System.out.println(color);//prints color of Dog class
8. System.out.println(super.color);//prints color of Animal class
9. }
10.}
11.class TestSuper1{
12.public static void main(String args[]){
13.Dog d=new Dog();
14.d.printColor();
15.}}
```

```
black
white
```

2) **super** can be used to invoke parent class method

The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

```
1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void eat(){System.out.println("eating bread...");}
6. void bark(){System.out.println("barking...");}
7. void work(){
8. super.eat();
9. bark();
10.}
11.}
12.class TestSuper2{
13.public static void main(String args[]){
14.Dog d=new Dog();
15.d.work();
16.}}
```

```
eating...
barking...
```

3) **super** is used to invoke parent class constructor.

The super keyword can also be used to invoke the parent class constructor. Let's see a simple example:

```
1. class Animal{
2. Animal(){System.out.println("animal is created");}
3. }
4. class Dog extends Animal{
5. Dog(){
6. super();
7. System.out.println("dog is created");
8. }
9. }
10. class TestSuper3{
11. public static void main(String args[]){
12. Dog d=new Dog();
13. }}
```

```
animal is created
dog is created
```

EJEMPLOS JAVA PROGRAMS

Java Programs | Java Programming Examples - **JavaTpoint**

DONDE APRENDER JAVA

1. Documentación de Oracle Java Documentation - Get Started (oracle.com)
2. Free learning Oracle Oracle Learning Explorer: Learn Oracle for Free | Oracle University.
3. **https://www.w3schools.com/java/default.asp**
4. **https://docs.oracle.com/javase/tutorial/**
5. **https://www.youtube.com/user/java**
6. https://www.sololearn.com/
7. https://www.tutorialspoint.com/java/index.htm
8. https://www.campusmvp.es/recursos/post/Descifrando-Java-lenguaje-plataforma-ediciones-implementaciones.aspx