

# Módulo 5

# Objetos y variables primitivas

Informática II



# Representación de Información

## Variables primitivas

- int
- char
- boolean
- double
- etc

## Objetos / Variables de tipo Referencia

- String
- Arreglos
- Objetos

# Valores iniciales de las variables

```
byte var1;    // initial value = 0
short var2;   // initial value = 0
int var3;     // initial value = 0
long var4;    // initial value = 0L
float var5;   // initial value = 0.0f
double var6;  // initial value = 0.0d
char var7;    // initial value = '\u0000'
Object var8;  // initial value = null
String var9;  // initial value = null
int[] var10;  // initial value = null
boolean flg;  // initial value = false
```

## Variables de tipo Referencia

Todas las variables de tipo referencia (Objetos, arreglos, Strings) comienzan con un valor inicial de **null**.

**¿Qué es null?**

**null** implica que el objeto todavía no contiene información.

## Variables de tipo referencia


A cualquier variable de tipo referencia se le puede hacer una asignación del valor **null** para indicar que la referencia aún no está inicializada.

Si intentamos acceder a algún método o variable de un objeto no inicializado, se generará la excepción **NullPointerException**.

```
public static void main(String[] args) {  
  
    Scanner s1 = null;  
    int x = s1.nextInt();  
  
}
```


```
run:  
[ ] Exception in thread "main" java.lang.NullPointerException  
    at primitive_reference.Primitive_Reference.main(Primitive\_Reference.java:21)  
Java Result: 1  
BUILD SUCCESSFUL (total time: 0 seconds)
```

```
byte var1;  
short var2;  
int var3;  
String var9;  
int[] var10;  
Object var11;
```



Nombre de Variable	Contenido	Dirección de memoria
byte var1	0	10F50
short var2	0	10F51
int var3	0	10F52
String var9	null	10F53
int[] var10	null	10F54
Object var11	null	10F55

```
var1 = 10;  
var2 = 3;  
var3 = 17;  
var9 = "Hola";
```



Nombre de Variable	Contenido	Dirección de memoria
byte var1	10	10F50
short var2	3	10F51
int var3	17	10F52
String var9	11A30	10F53
	...	
	...	
var9[0]	H	11A30
var9[1]	o	11A31
var9[2]	l	11A32
var9[3]	a	11A33
var9[4]	end of String	11A34

```
var10 = new int[3];  
var11 = new Object();
```



Nombre de Variable	Contenido	Dirección de memoria
int[] var10	12A00	10F54
Object var11	14FF1	10F55
	...	
var10[0]	0	12A00
var10[1]	0	12A01
var10[2]	0	12A02
	...	
Object.a	0	14FF1
Object.b	FALSE	14FF2
Object.c	0	14FF3

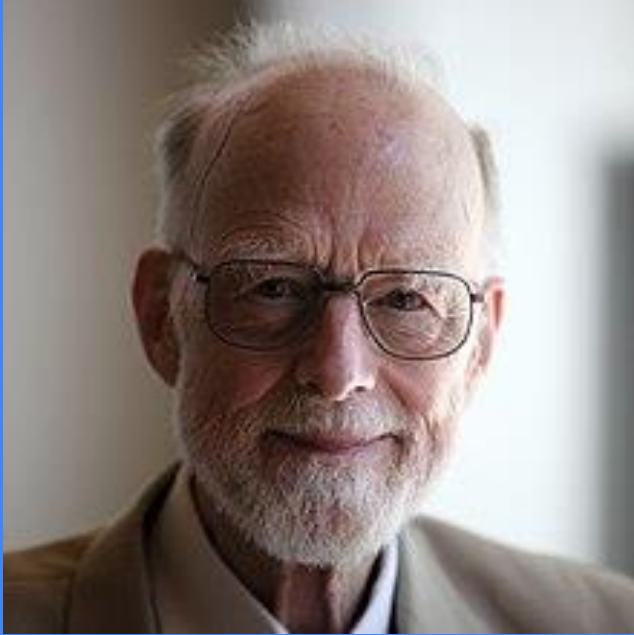
La diferencia entre una variable primitiva y una variable de tipo referencia es la manera en la que se almacena la información.

Las variables primitivas **almacenan directamente la información** (y tiene un tamaño predefinido)

Las variables de tipo referencia **almacenan un apuntador a la localidad de memoria** en donde se almacenará la información, y tienen un tamaño indefinido hasta la ejecución.

Nombre de Variable	Contenido	Dirección de memoria
byte var1	10	10F50
short var2	3	10F51
int var3	17	10F52
String var9	11A30	10F53
int[] var10	12A00	10F54
Object var11	14FF1	10F55
	...	
	...	
	...	
	...	
var9[0]	H	11A30
var9[1]	o	11A31
var9[2]	l	11A32
var9[3]	a	11A33
var9[4]	end of String	11A34
	...	
var10[0]	0	12A00
var10[1]	0	12A01
var10[2]	0	12A02
	...	
Object.a	0	14FF1
Object.b	FALSE	14FF2
Object.c	0	14FF3





*"Null references were created in 1964 -  
how much have they cost? (...)*

*This has led to innumerable errors,  
vulnerabilities, and system crashes,  
which have probably caused a billion  
dollars of pain and damage in the last  
forty years."*

Sir Charles Anthony Richard Hoare

# Variables de tipo referencia

Instanciar una variable con un valor vacío es distinto a inicializarlo con un valor de tipo nulo.

`null` es una palabra reservada.

```
int[] c1 = null;  
int[] c2 = new int[0];
```

```
if (c1 == c2) {  
    System.out.println("Iguales");  
} else {  
    System.out.println("Diferentes");  
}
```

run:

Diferentes

BUILD SUCCESSFUL (total time: 0 seconds)

|

# Variable de tipo referencia

Es importante implementar validaciones de este tipo cuando utilizamos métodos.

Veamos los siguientes ejemplos:

```
public static void main(String[] args) {  
  
    int[] x = null;  
    printArray(x);  
}  
  
public static void printArray(int[] array) {  
    for(int i = 0; i<array.length; i++) {  
        System.out.println(array[i]);  
    }  
}
```

Al momento de intentar acceder a la variable **length** del objeto **array**, se levantará la excepción **NullPointerException**!

```
run:  
[x] Exception in thread "main" java.lang.NullPointerException  
    at primitive_reference.Primitive_Reference.printArray(Primitive_Reference.java:25)  
    at primitive_reference.Primitive_Reference.main(Primitive_Reference.java:21)  
Java Result: 1  
BUILD SUCCESSFUL (total time: 0 seconds)
```

# Variable de tipo referencia

¿Cómo podemos evitarlo?

Respuesta: Incluyendo una validación de la validez del arreglo recibido.

```
public static void main(String[] args) {  
  
    int[] x = null;  
    printArray(x);  
}  
  
public static void printArray(int[] array) {  
  
    //check pointer validity  
    if (array == null){  
        System.out.println("Imposible imprimir.");  
        return;  
    }  
  
    for(int i = 0; i<array.length; i++) {  
        System.out.println(array[i]);  
    }  
}
```

## Programación a la defensiva!

# Notación UML

# Notación UML

El **Unified Model Language** es una serie de estándares con el objetivo de unificar la forma en la que se modela (y diseña) el software.

<<Java Class>>

 **Animal**

animal

- name: String
- race: String
- foods: String[]
- hunger: int

•<sup>c</sup> Animal(String, String, String[], int)

•<sup>c</sup> Animal()

• eat(String): void

• getHunger(): int

• setHunger(int): void

```
public class Animal2 {  
  
    public String name;  
    public String race;  
    public String[] foods;  
    private int hunger;  
  
    public Animal(String name, String race,  
                   String[] foods, int hunger) {}  
  
    public Animal() {}  
  
    public void eat(String inputFood) {}  
  
    public int getHunger() {}  
  
    public void setHunger(int hunger) {}  
  
}
```

Tipo de Archivo (opcional)

Nombre de la clase

Paquete (opcional)

<<Java Class>>

Ⓜ **Animal**

animal



## Lista de los atributos

```
• name: String  
• race: String  
• foods: String[]  
▪ hunger: int
```

1. Modificador de acceso
2. Nombre de la variable
3. : (dos puntos)
4. Tipo de datos



Para los modificadores de acceso, se puede utilizar la nomenclatura:

- + Elementos públicos
- Elementos privados

## Lista de métodos

```
• Animal(String,String,String[],int)  
• Animal()  
• eat(String):void  
• getHunger():int  
• setHunger(int):void
```

1. Modificador de acceso
2. Nombre del método
3. Lista de parámetros de entrada
4. : Dos puntos
5. Valor de retorno



1. Cuando el método retorne un valor void, se puede omitir el valor de retorno.
2. También se puede incluir los nombres del parámetro de entrada en el formato:  
**nombre: tipo de datos**

# Ejercicio

## Clase de Estudiantes

Queremos representar un salón de clases de estudiantes de preparatoria.

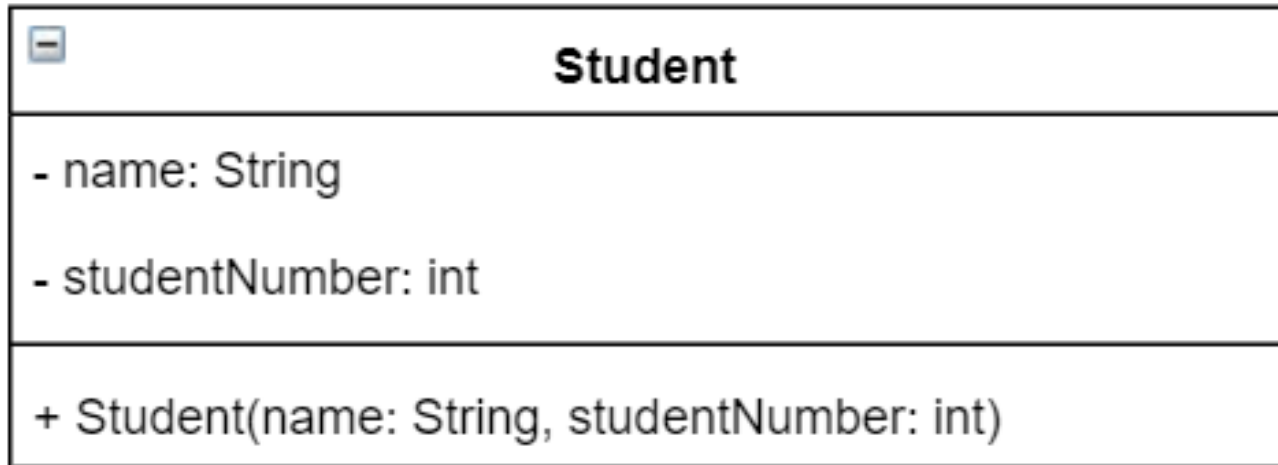
Cada estudiante deberá poder almacenar nombre y número de estudiante.

Cada salón de clases deberá poder almacenar un arreglo con los estudiantes inscritos, y el número de salón en donde se lleva a cabo la clase.

## Ejercicio: Salón de Clases

Diseñamos una primera clase Student que pueda almacenar:

- Nombre
- Número de estudiante

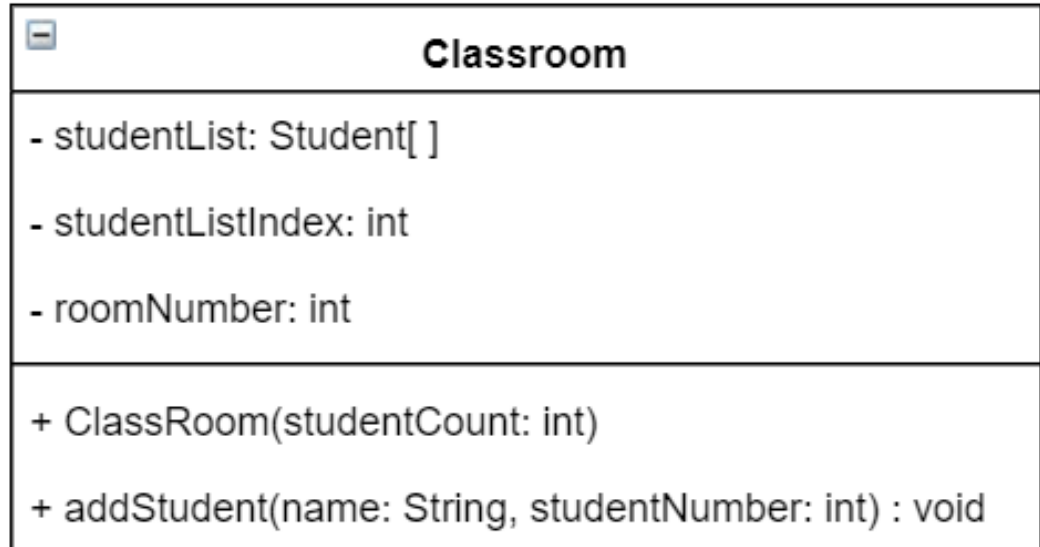


```
public class Student {  
    private String name;  
    private int studentNumber;  
  
    public Student(String name, int studentNumber) {  
        this.name = name;  
        this.studentNumber = studentNumber;  
    }  
}
```

## Ejercicio: Salón de Clases

Posteriormente diseñamos una segunda clase Classroom que pueda almacenar:

- Lista de Estudiantes
- Índice del último estudiante registrado en el arreglo
- Número de salón de clases



## Ejercicio: Salón de Clases

```
public class Classroom{

    private Student[] studentList;
    private int studentListIndex;
    private int roomNumber;

    public Classroom(int studentCount, int roomNumber) {

        if (studentCount <= 0 || roomNumber < 0) {
            System.out.println("Error, studentCount (" + studentCount + ") invalido.");
            System.exit(0);
        }

        this.studentList = new Student[studentCount];
        this.studentListIndex = 0;
        this.roomNumber = roomNumber;
    }

    public void addStudent(String name, int studentNumber){
        //only insert student into the list if list has space for new student
        if (this.studentListIndex < this.studentList.length ) {
            this.studentList[this.studentListIndex] = new Student(name, studentNumber);
            this.studentListIndex++;
        } else {
            System.out.println("No hay espacio para " + name + "!");
        }
    }

}
```

## Casos de Prueba

```
public static void main(String[] args) {  
  
    Classroom informatica = new Classroom(5, 123);  
    informatica.addStudent("Omar", 1234);  
    informatica.addStudent("Jose", 1234);  
    informatica.addStudent("Eduardo", 1234);  
    informatica.addStudent("Marco", 1234);  
    informatica.addStudent("Santiago", 1234);  
    informatica.addStudent("Pepe", 1234); //No hay espacio para Pepe!  
  
    Classroom matematicas = new Classroom(-5, 944); //Errorr!  
  
}
```

### OUTPUT

```
No hay espacio para Pepe!  
Error, studentCount (-5) invalido
```