



1.

# Encapsulación

Tipos de datos, clases y objetos

# Tipos de datos

- Un programa se puede entender como una serie de instrucciones que operan sobre un conjunto de datos (**entrada**) y generan otro conjunto de datos (**salida**)
- Todos los lenguajes de programación utilizados en la actualidad tienen instrucciones de selección (**tipo if**) y de repetición (**tipo while o for**), de forma que es posible crear cualquier programa (teorema de Jacopini)
- Los tipos de datos disponibles en un lenguaje de programación limitan enormemente las posibilidades a la hora de crear programas si no es posible representar un tipo de datos concreto, no se podrá operar sobre ellos

# Tipos de datos

- Ejemplo: si un lenguaje de programación no dispone de datos de tipo decimal, no podrá crear un programa que calcule el área de un círculo
- ¿Qué tipos de datos son los que se usan en un programa?

El juego tiene lugar sobre un tablero en el que se representa un mapa del mundo. En el mapa se dispondrán 6 continentes que se identificarán con un color: Asia (CYAN), África (VERDE), Europa (AMARILLO), América del Norte (VIOLETA), América del Sur (ROJO) y Australia (AZUL).

Las casillas son los países de los que consta cada continente. Es decir, los países se representarán como casillas cuadradas o rectangulares para facilitar su pintado.

Las casillas en NEGRO se considerarán océanos o mares.

# Tipos de datos

- Si el lenguaje de programación que vamos a usar es C

Tipo	Rango
char	-127 a 127
unsigned char	0 a 255
signed char	-127 a 127
int	-32767 a 32767
unsigned int	0 a 65535
signed int	-32767 a 32767
short int	-32767 a 32767
unsigned short int	0 a 65535
long int	-2147483647 a 214783647
signed long int	-2147483647 a 214783647
unsigned long int	0 a 4294967295
float	6 dígitos de precisión
double	10 dígitos de precisión
long double	10 dígitos de precisión

- Si el lenguaje de programación que vamos a usar es Python

Tipo	Notas
int	Número entero
float	Coma flotante
bool	Valor verdadero o falso
str	Inmutable
list	Mutable
tuple	Inmutable
set	Mutable, sin orden, sin duplicados
frozenset	Inmutable, sin orden, sin duplicados

# Tipos de datos

- Opción 1 (en lenguaje C):

```
char color_continente[8];  
char colores[6][8]= { "AMARILLO", "AZUL", "CYAN", "ROJO", "VERDE", "VIOLETA" };  
char nombre_continente[20];  
char continentes[6][20]= { "África", "América del Norte", "América del Sur", "Asia", "Australia", "Europa" };  
char nombre_pais[15];  
char[42][15] paises= { "Perú", "Brasil", "Argentina", "Venezuela", ... };
```

- Es necesario definir un conjunto de funciones para asociar los países y los colores a los continentes
- Cada función del programa tiene que definir las variables *pais* y *nombre\_continente*

# Tipos de datos

- ¿Qué tipos de datos son los que se usan en un programa?

El juego tiene lugar sobre un tablero en el que se representa un **mapa** del mundo. En el **mapa** se dispondrán 6 **continentes** que se identificarán con un **color**: Asia (CYAN), África (VERDE), Europa (AMARILLO), América del Norte (VIOLETA), América del Sur (ROJO) y Australia (AZUL).

Las **casillas** son los **países** de los que consta cada **continente**. Es decir, los **países** se representarán como **casillas** cuadradas o rectangulares para facilitar su pintado.

Las **casillas** en NEGRO se considerarán océanos o mares.

Los tipos de datos que necesitamos en el programa **no** son tipos de datos que están disponibles en **C**.

# Tipos de datos

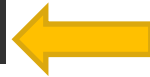
- Opción 2 (en lenguaje C):

```
struct Continente {  
    char nombre_continente;  
    char color_continente[8];  
    Pais paises[12];  
};  
  
struct Pais {  
    char nombre_pais[15];  
    Continente continente;  
};  
  
int main()  
{  
    Struct Continente Asia;  
    Struct Continente Australia;  
  
    Australia.color_continente= "JADE";
```

Se definen nuevos tipos de datos a través de estructuras, facilitando enormemente la creación del programa

## **Problema:** Integridad de los datos

No es posible restringir el valor que toman las variables en las distintas funciones del programa



“JADE” no es un valor correcto para el color de los continentes

# Catástrofes

## ARIANE 5

### Flight 501 Failure

Report by the Inquiry Board

The Chairman of the Board :

Prof. J. L. LIONS

[originally appeared at  
<http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html>]

### FOREWORD

On 4 June 1996, the maiden flight of the Ariane 5 launcher ended in a failure. Only about 40 seconds after initiation of the flight sequence, at an altitude of about 3700 m, the launcher veered off its flight path, broke up and exploded. Engineers from the Ariane 5 project teams of CNES and Industry immediately started to investigate the failure. Over the following days, the Director General of ESA and the Chairman of CNES set up an independent Inquiry Board and nominated the following members :



- The internal SRI software exception was caused during execution of a data conversion from 64-bit floating point to 16-bit signed integer value. The floating point number which was converted had a value greater than what could be represented by a 16-bit signed integer. This resulted in an Operand Error. The data conversion instructions (in Ada code) were not protected from causing an Operand Error, although other conversions of comparable variables in the same place in the code were protected.



# Catástrofes

## Mars Climate Orbiter

### Mishap Investigation Board

#### Phase I Report

November 10, 1999

#### 4. Mars Climate Orbiter (MCO) Root Cause and Mars Polar Lander (MPL) Recommendations

During the mishap investigation process, specific policy is in-place to conduct the investigation and to provide key definitions to guide the investigation. NASA Procedures and Guidelines (NPG) 8621 Draft 1, "NASA Procedures and Guidelines for Mishap Reporting, Investigating, and Recordkeeping" provides these key definitions for NASA mishap investigations. NPG 8621 (Draft 1) defines a root cause as: "Along a chain of events leading to a mishap, the first causal action or failure to act that could have been controlled systematically either by policy/practice/procedure or individual adherence to policy/practice/procedure". Based on this definition, the Board determined that there was one root cause for the MCO mishap.

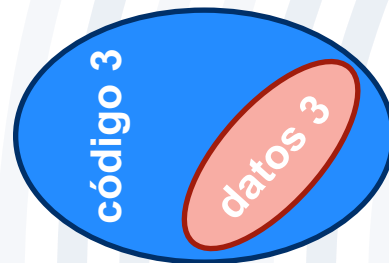
##### MCO Root Cause

The MCO MIB has determined that the root cause for the loss of the MCO spacecraft was the failure to use metric units in the coding of a ground software file, "Small Forces," used in trajectory models. Specifically, thruster performance data in English units instead of metric units was used in the software application code titled SM\_FORCES (small forces). The output from the SM\_FORCES application code as required by a MSOP Project Software Interface Specification (SIS) was to be in metric units of Newton-seconds (N-s). Instead, the data was reported in English units of pound-seconds (lbf-s).

The Angular Momentum Desaturation (AMD) file contained the output data from the SM\_FORCES software. The SIS, which was not followed, defines both the format and units of the AMD file generated by ground-based computers. Subsequent processing of the data from AMD file by the navigation software algorithm therefore, underestimated the effect on the spacecraft trajectory by a factor of 4.45, which is the required conversion factor from force in pounds to Newtons. An erroneous trajectory was computed using this incorrect data.

# Tipos de datos: Encapsulación

- Una de las principales características de la Programación Orientada a Objetos es la **encapsulación**
  - Se asegura la integridad de los datos, **limitando** los trozos de código del programa que pueden acceder a unos datos dados
  - Idealmente no existe ningún trozo de código que pueda acceder a todos los datos del programa
  - Hace uso del **principio de ocultación**, en virtud del cual solo un trozo de código puede “ver” un conjunto de datos dado



# Clases

- En Programación Orientada a Objetos los tipos de datos son las **entidades** que se quieren representar en el programa
  - Si necesitamos representar la **edad** de una persona esa entidad será un entero
  - Si necesitamos representar un **continente**, entonces la entidad será el continente y tendrá una serie de atributos (o variables) que lo definen y que, a su vez, tienen un tipo de dato asociado
    - El color del continente es una **cadena de texto**
    - Los países del continente son un array de tipo de **Pais**

# Clases

- En Programación Orientada a Objetos estas entidades del programa, es decir, los tipos de datos se conceptualizan como **clases**
- En Programación Orientada a Objetos **todos los tipos de datos son clases** y los valores concretos de esos tipos de datos, es decir, de las clases, se denominan **objetos**
  - Asia es un objeto que tiene como tipo de dato Continente
  - Venezuela es un objeto que tiene como tipo de dato Pais
  - Amarillo es un objeto que tiene como tipo de datos una cadena de texto

# Clases en Java

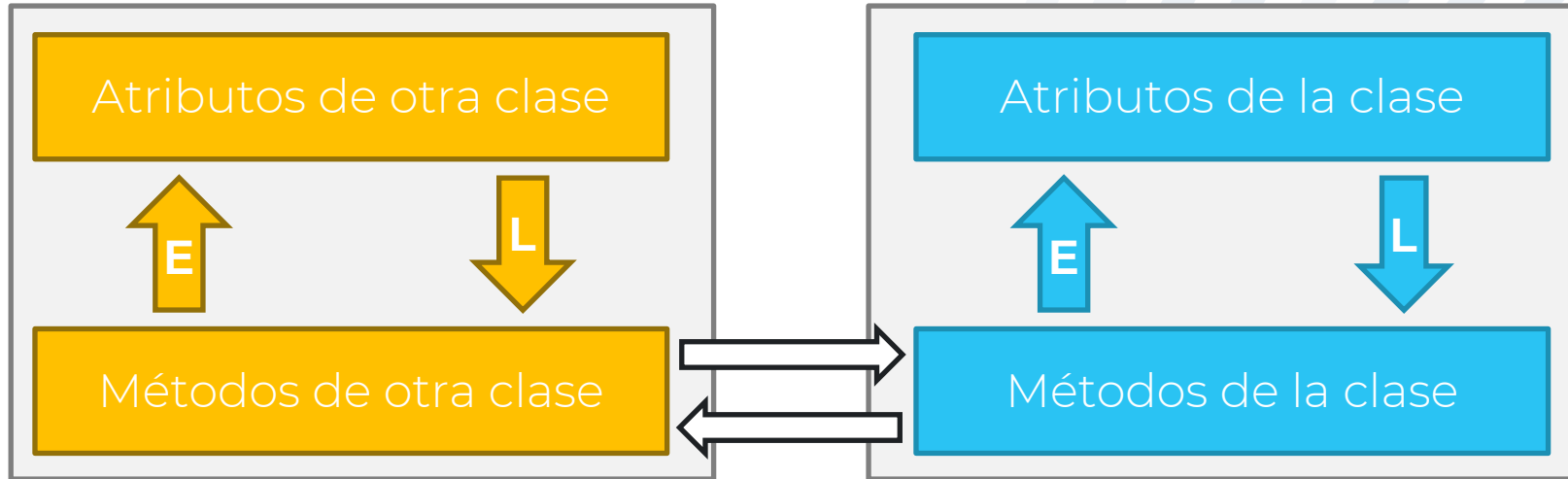
- Java es un lenguaje de programación que está basado en el paradigma de la Programación Orientada a Objetos que distingue entre dos especies de tipos de datos
  - **Tipos de datos primitivos**, son los tipos de datos que están vinculados a la representación de los datos en el computador. Son los **mismos tipos de datos que los del lenguaje C**
  - **Clases**, son tipos de datos que están vinculados a las entidades de alto nivel del programa y, por exclusión, son tipos de datos que no se pueden representar de forma directa en el computador

# Clases: Encapsulación

- Con carácter general, las clases **contienen** dos tipos de código
  - **Las variables o atributos** que son la características que definen la entidad representada por la clase
    - **Ejemplo:** el nombre del continente, el conjunto de países que forman parte del continente, el conjunto de continentes con los que limita
  - **Las funciones o métodos** que acceden a los atributos para permitir su lectura y escritura, y para proporcionar la funcionalidad requerida en el programa
    - **Ejemplo:** la función que obtiene el conjunto de países que son frontera de un continente

# Clases: Encapsulación

- La idea para resolver el problema de la integridad de los datos es que las **únicas funciones** que pueden acceder a los atributos, tanto para leerlos como para escribir sobre ellos, son las funciones de la clase



# Clases: Estructura

```
public [final | abstract] class <nombre_clase>
```

```
{
```

```
// Atributos
```

```
<tipo_acceso> <tipo> <nombre_atrib>;
```

```
<tipo_acceso> <static | final> <tipo> <nombre_atrib> = <valor>;
```

```
// Constructores
```

```
public <nombre_clase> (<tipo> <nombre> *) { <código> }
```

```
// Métodos de lectura (getters)
```

```
public <tipo_atributo> get<nombre_atributo> ( ) {
```

```
    return <nombre_atributo>;
```

```
}
```

```
// Métodos de escritura (setters)
```

```
public void set<nombre_atributo> (<tipo_atributo> <nombre> * ) { <código> }
```

```
// Otros métodos
```

```
<tipo_acceso> <static | final> <tipo> <nombre_método> (<tipo> <nombre> *) {
```

```
    <código>
```

```
}
```

```
}
```

Declaración de los atributos (variables) de la entidad

Reserva memoria para los atributos de la entidad

Funciones que obtienen los valores de los atributos de la entidad

Funciones que escriben los valores de los atributos

Funciones que operan sobre los atributos para obtener las funcionalidades del programa



# Clases en Java: <tipo\_acceso>

- Se dispone de **cuatro tipos de acceso**, que se pueden aplicar tanto a los atributos como a los métodos
  - **Público** (*public*)
    - **Métodos públicos**, pueden ser invocados desde cualquier método de cualquier clase del programa
    - **Atributos públicos**, pueden ser leídos o escritos desde cualquier método de cualquier clase del programa

```
public class RiskETSE {  
    public static void main(String[] args) {  
        Continente australia= new Continente();  
        System.out.println("Nombre del continente " + australia.nombre);  
        australia.nombre= "Pon cualquier nombre";  
    }  
}
```

# Clases en Java: <tipo\_acceso>

- Se dispone de **cuatro tipos de acceso**, que se pueden aplicar tanto a los atributos como a los métodos
  - **Privado** (*private*)
    - **Métodos privados**, solamente pueden ser invocados desde cualquier método de la clase a la que pertenecen
    - **Atributos privados**, solamente se pueden leer o escribir desde los métodos de la clase a la que pertenecen

```
public class RiskETSE {  
    public static void main(String[] args) {  
        Continente a = new Continente();  
        System.out.println("El nombre del continente es: " + a.nombre);  
        a.nombre = "Pon cualquier nombre";  
    }  
}
```

# Clases en Java: <tipo\_acceso>

- Se dispone de **cuatro tipos de acceso**, que se pueden aplicar tanto a los atributos como a los métodos
  - **Privado** (*private*)
    - Para leer y escribir los atributos privados de una clase se usan dos tipos de métodos: **getters** y **setters**, respectivamente

```
public class RiskETSE {  
    public static void main(String[] args) {  
        Continente australia= new Continente();  
        System.out.println("Nombre del continente " + australia.getNombre());  
        australia.setNombre("Pon cualquier nombre");  
    }  
}
```

getNombre() obtiene el nombre del continente

setNombre(String nombre) asigna el valor del argumento

# Clases en Java: <tipo\_acceso>

- Se dispone de **cuatro tipos de acceso**, que se pueden aplicar tanto a los atributos como a los métodos
  - **Acceso a paquete** (□)
    - **Métodos de acceso a paquete**, solamente pueden invocarlos los métodos de las clases que pertenecen al mismo paquete de la clase en la que se encuentra el método. Se comportan como privados para cualquier otro método
    - **Atributos de acceso a paquete**, pueden ser leídos y escritos directamente por los métodos de las clases que pertenecen al mismo paquete de la clase de los atributos. Se comportan como privados para cualquier otro método

# Clases en Java: <tipo\_acceso>

- Se dispone de **cuatro tipos de acceso**, que se pueden aplicar tanto a los atributos como a los métodos

- **Acceso a paquete** (□)

```
package risketse;  
  
import java.util.ArrayList;  
  
public class Continente {  
    int numeroEjercitos;  
    String nombre;  
    ArrayList<Pais> paises;  
    private String color;  
}
```

Los atributos **numeroEjercitos**, **nombre** y **paises** tienen acceso a paquete

Todos los métodos de las clases que no pertenecen al **paquete risketse** deberán utilizar un método de la clase **Continente** para acceder a dichos atributos

# Clases en Java: <tipo\_acceso>

- Se dispone de **cuatro tipos de acceso**, que se pueden aplicar tanto a los atributos como a los métodos
  - **Acceso protegido** (*protected*)
    - **Métodos protegidos**, pueden ser invocados por los métodos de las clases que pertenecen al mismo paquete de la clase y por los métodos de las subclases de la clase
    - **Atributos protegidos**, pueden ser leídos y escritos por los métodos de las clases que pertenecen al mismo paquete de la clase y por los métodos de las subclases de la clase

Se introducen para facilitar la **herencia de atributos** entre clases que se encuentran en paquetes diferentes

# Clases en Java: <tipo\_acceso>

- Resumen de los **niveles de acceso** a los métodos y atributos en función de la clase en la que se encuentran los métodos que acceden a esos métodos o atributos

Modificador	Clase	Paquete	Subclase	Resto
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
□	✓	✓	✗	✗
private	✓	✗	✗	✗

# Clases en Java: <tipo\_acceso>

- Buenas prácticas de programación (I)

Por lo general, los atributos de una clase deben ser siempre privados, para que no puedan ser modificados de forma incorrecta, y los métodos, públicos, para que se pueda acceder a los atributos de las clases



- Buenas prácticas de programación (II)

Se pueden usar métodos privados para facilitar la codificación de los métodos





# Clases: Getters y setters

- Los **getters** son métodos que **devuelven el valor** que tienen los atributos en cada momento
  - Cada atributo de la clase tiene un **único** getter
  - El nombre del getter es **get<nombre\_atributo>**, donde la primera de <nombre\_atributo> es en mayúscula
  - Típicamente los getters únicamente contienen el código asociado a devolución del valor del atributo (**return <valor\_atributo>**)

```
public ArrayList<Pais> getPaises() { return países; }
```

```
public String getNombre() { return nombre; }
```

```
public String getColor() { return color; }
```

# Clases: Getters y setters

- Los **setters** son métodos que **escriben el valor** de los atributos
  - Cada atributo de la clase tiene un **único** setter
  - El nombre del getter es **set<nombre\_atributo>**, donde la primera de <nombre\_atributo> es en mayúscula
  - Típicamente un setter tiene código sobre las condiciones que debe de cumplir el argumento para que sea un valor correcto del atributo, manteniendo así la integridad de los datos

```
public void setColor(String color) {  
    if(color.equals("AMARILLO") || color.equals("AZUL") || color.equals("CYAN") ||  
        color.equals("ROJO") || color.equals("VERDE") || color.equals("VIOLETA"))  
        this.color= color;  
    else System.out.println("Color no permitido");  
}
```

# Clases: Getters y setters

```
public class Continente {  
    private int numeroEjercitos;  
    private String nombre;  
    private ArrayList<Pais> paises;  
    private String color;  
  
    public String getNombre() {  
        return nombre;  
    }  
  
    public void setNombre(String nombreContinente) {  
        if(nombre.equals("África") ||  
            nombre.equals("América del Norte") ||  
            nombre.equals("América del Sur") ||  
            nombre.equals("Asia") ||  
            nombre.equals("Australia") ||  
            nombre.equals("Europa"))  
            this.nombre= nombreContinente;  
        else  
            System.out.println("No existe el continente");  
    }  
}
```

Tipo de dato primitivo: **int**

Tipos de datos asociados a las entidades del programa: **String** y **ArrayList**

Función que obtiene el valor del tipo de dato asociado a la variable **nombre**

Función que escribe el valor del atributo **nombre** con el valor de la variable que se pasa como argumento (**nombreContinente**). Esta función limita los valores del atributo **nombre** a los nombres de los seis continentes que forman parte del mapa del programa

# Clases: Getters y setters

- Buenas prácticas de programación (III)

Los setters siempre deben incluir una condición para evitar la escritura de **null** como valor de los atributos



- Buenas prácticas de programación (IV)

Por lo general, todos los atributos deben tener setters y getters, salvo que solamente tengan interés o sean usados como parte de la programación de los métodos de la clase



# Clases: Constructores

- La declaración de un atributo de una clase **tiene distintos efectos** en función de su tipo de dato
- Si el atributo es un tipo de dato primitivo
  - En la declaración del atributo se **realiza la reserva de memoria** y se le asigna un valor inicial de acuerdo con la siguiente tabla

Tipo	Valor inicial
boolean	False
char	'\u0000'
byte, short, int, long	0
float	+0.0f
double	+0.0d

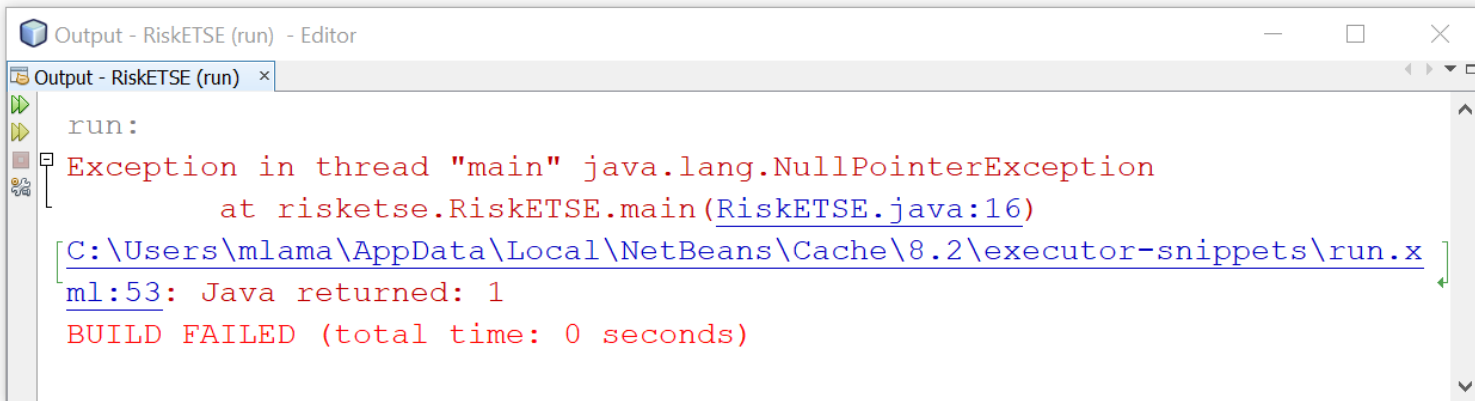
# Clases: Constructores

- La declaración de un atributo de una clase **tiene distintos efectos** en función de su tipo de dato
- **Si el atributo es de tipo clase**
  - No se reserva automáticamente memoria para el atributo, ya que se desconoce cuánto ocupará en la memoria del computador
  - Se le asigna en todos los casos un valor inicial de **null**, de manera que en la práctica se trata como una variable que no existe, generando un error tipo **NullPointerException** si se intenta usar
  - La reserva de memoria se tiene que hacer manualmente para cada atributo o variables de tipo clase y para ello se **invocan los constructores** a través del **operador new**

# Clases: Constructores

Como el nombre de Continente se inicializa a null, la comparación es: `null.equals("Australia")`, generando un error

```
public class RiskETSE {  
    public static void main(String[] args) {  
        // Continente() asigna valores iniciales por defecto a los atributos  
        Continente australia= new Continente();  
        if(!australia.getNombre().equals("Australia"))  
            australia.setNombre("Australia");  
        System.out.println("Nombre del continente: " + australia.getNombre());  
    }  
}
```



The screenshot shows the 'Output - RiskETSE (run) - Editor' window. It displays the following text:

```
run:  
Exception in thread "main" java.lang.NullPointerException  
    at risketse.RiskETSE.main(RiskETSE.java:16)  
[C:\Users\mlama\AppData\Local\NetBeans\Cache\8.2\executor-snippets\run.x  
ml:53: Java returned: 1  
BUILD FAILED (total time: 0 seconds)
```

# Clases: Constructores

- De la misma forma, al declarar una variable que tiene como tipo de dato una clase, es decir, al declarar un objeto, **no se reserva automáticamente memoria** para dicho objeto
  - Si una variable no se inicializa o no se reserva memoria, **se genera un error de compilación**

```
public class RiskETSE {  
    public static void main(String[] args) {  
        Continente australia;  
  
        if (!australia.getNombre().equals("Australia"))  
            australia.setNombre("Australia");  
        System.out.println("Nombre del continente: " + australia.getNombre());  
    }  
}
```

variable australia might not have been initialized  
----  
(Alt-Enter shows hints)



# Clases: Constructores

- Un constructor es un método/función que se invoca para
  - reservar memoria para un **objeto** de la clase,
  - reservar memoria para los atributos de dicho objeto y
  - asignar valores iniciales a los atributos del objeto

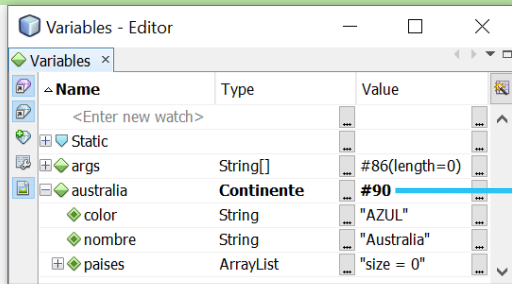
Un constructor no devuelve ningún tipo de dato y su nombre coincide con el nombre de la clase a la que pertenece

```
public Continente(ArrayList<Pais> paises, String nombre, String color) {  
    for(Pais pais : paises) pais.setContinente(this);  
    this.paises = paises;  
    this.nombre = nombre;  
    this.color= color;  
}
```

# Clases: Constructores

- Un constructor se invoca una **única** vez para cada objeto
  - En realidad cuando se reserva memoria para un objeto, el nombre del objeto es una **referencia a la posición de memoria** en la que se almacenan los datos del objeto

```
public class RiskETSE {  
    public static void main(String[] args) {  
        Continente australia= null;  
        australia= new Continente("Australia", "AZUL");  
        australia= new Continente("Oceanía", "AZUL");  
    }  
}
```



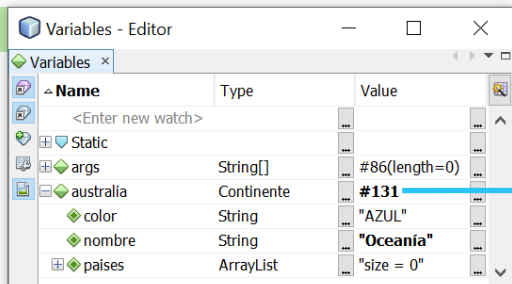
Name	Type	Value
<Enter new watch>		
Static		
args	String[]	#86(length=0)
australia	Continente	#90
color	String	"AZUL"
nombre	String	"Australia"
países	ArrayList	"size = 0"

australia es una referencia a la posición de memoria #90

# Clases: Constructores

- Un constructor se invoca una **única** vez para cada objeto
  - Una vez se ha creado la referencia y se ha invocado al constructor para reservar memoria, si se vuelve a invocar al constructor, se **apuntará a una nueva posición de memoria**

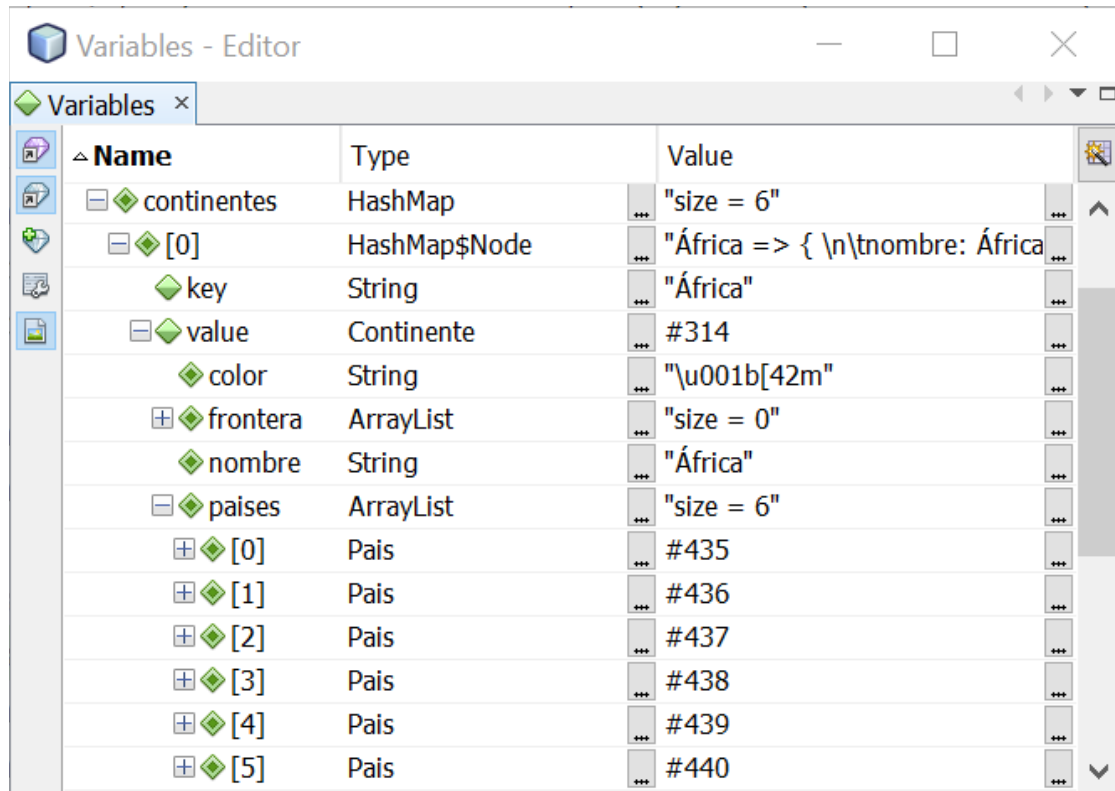
```
public class RiskETSE {  
    public static void main(String[] args) {  
        Continente australia= null;  
        australia= new Continente("Australia", "AZUL");  
        australia= new Continente("Oceanía", "AZUL");  
    }  
}
```



Name	Type	Value
<Enter new watch>		
Static		
args	String[]	#86(length=0)
australia	Continente	#131
color	String	"AZUL"
nombre	String	"Oceania"
países	ArrayList	"size = 0"

australia es una referencia a la nueva posición de memoria #131

# Clases: Constructores



Name	Type	Value
continentes	HashMap	"size = 6"
[0]	HashMap\$Node	"África => { \n\tnombre: África"
key	String	"África"
value	Continente	#314
color	String	"\u001b[42m"
frontera	ArrayList	"size = 0"
nombre	String	"África"
países	ArrayList	"size = 6"
[0]	Pais	#435
[1]	Pais	#436
[2]	Pais	#437
[3]	Pais	#438
[4]	Pais	#439
[5]	Pais	#440

Desde la posición de memoria de un objeto (#314, continente con nombre África) **se hace referencia** a posiciones de memoria de aquellos atributos de tipo objeto (#435 a #440)

Los valores de atributos que son de tipo primitivo **se almacenan** en la memoria reservada para el objeto

# Clases: Constructor por defecto

- El constructor por defecto es un **constructor que proporciona automáticamente Java** cuando no se especifica ningún otro constructor en la clase
  - No tiene argumentos
  - Inicializa los atributos a sus **valores por defecto**, de modo que los atributos de tipo clase tomarán el valor **null**
  - Tiene el mismo efecto que especificar un constructor que **no tiene argumentos y con el cuerpo vacío**, es decir, sin ningún tipo de código

```
public Continente() {  
    // No tiene ningún tipo de código  
}
```

# Clases: Constructores

- Buenas prácticas de programación (V)

No es buena práctica de programación no definir constructores y usar el constructor por defecto proporcionado por Java para reservar memoria para los objetos



- Buenas prácticas de programación (VI)

Se deben proporcionar constructores que tienen como argumentos los valores de los atributos que es obligatorio inicializar para crear un objeto



# Clases: Métodos funcionales

- Una vez han sido definidos los atributos que caracterizan a una clase, se identifican y codifican los métodos que pertenecen a dicha clase y que realizan las operaciones que necesitan otras clases para implementar la **funcionalidad** del programa
- Ejemplos de métodos funcionales que podrían pertenecer a la clase **Continente**
  - **boolean esPaisDelContinente(Pais pais)**, método que indica si un país dado pertenece o no al continente
  - **void obtenerFrontera()**, obtiene automáticamente los países que son frontera de un continente, es decir, los países que son frontera con otros países que pertenecen a otros continentes

# Clases: Métodos funcionales

```
public class Continente {  
    private String nombre;  
    private ArrayList<Pais> paises;  
    private String color;  
  
    public boolean esPaisDelContinente(Pais pais) {  
        return paises.contains(pais);  
    }  
  
    public ArrayList<Pais> obtenerFrontera() {  
        ArrayList<Pais> fronteraCon= new ArrayList<>();  
        for(Pais paisCon : paises)  
            for(Pais paisFrontera : paisCon.getFrontera())  
                if(!esPaisDelContinente(paisFrontera) &&  
                    !fronteraCon.contains(paisFrontera))  
                    fronteraCon.add(paisFrontera);  
        return fronteraCon;  
    }  
}
```

Los métodos funcionales acceden **directamente** a los valores de los atributos

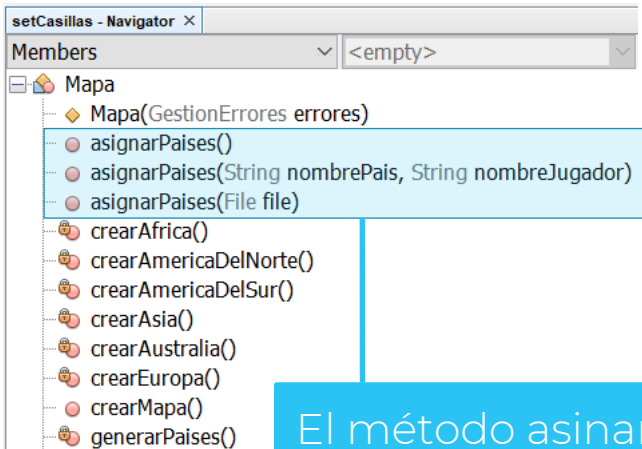
Tiene sentido crear este tipo de métodos en una clase cuando los **datos necesarios para realizar las operaciones** son los valores de los atributos de dicha clase

Los métodos se invocan **desde los objetos de las clases** con el operador “.”



# Clases: Métodos funcionales

- Un método o un constructor de una clase puede **tener varias implementaciones**, lo que otorga una mayor flexibilidad a la hora de usar la clase en diferentes contextos
- En este caso se dice que el método está **sobrecargado**



- El nombre del método **es el mismo** en todas las implementaciones
- Los tipos de los argumentos **deben ser diferentes**, ya que en caso contrario el compilador no podría distinguir cuál de las implementaciones debería invocar

El método asinarPaises tiene tres implementaciones diferentes

# Clases: Métodos funcionales

- No hay una regla fija que indique **cuántas** implementaciones **diferentes** debe tener un método en una misma clase y en la mayoría de los casos depende del contexto de invocación de los métodos
  - El objetivo del método **tiene que ser el mismo** en todas las implementaciones (por ejemplo, asignar países)
  - Se desaconseja el **uso de condiciones sobre los argumentos** con el fin de seleccionar el trozo de código que se deberá de ejecutar en cada caso
- La sobrecarga de constructores es mucho **más habitual**, ya tienen diferentes argumentos en función de la disponibilidad de los valores de los atributos a la hora de crear la clase

# Clases: Métodos funcionales

- (Peores) **alternativas** a la sobrecarga de métodos para asignarPaises

1

```
public void asignarPaises(String nomPais, String nomJugador, File file) {  
    if(nomPais!=null && nomJugador!=null && file==null) {  
        /* Código 1 */  
    } else if(nomPais==null && nomJugador==null && file!=null) {  
        /* Código 2 */  
    } else if(nomPais==null && nomJugador==null && file==null) {  
        /* Código 3 */  
    }  
}
```

2

```
public void asignarPaisesPorNombre(String nomPais, String nomJugador)  
  
public void asignarPaisesDesdeFichero(File file)  
  
public void asignarPaises()
```

# Clases en Java: Método toString

- toString es un método que devuelve la **representación en texto** de un objeto de una clase

```
@Override  
public String toString()
```

- toString es un método que tienen **todas las clases de Java**, tanto las propias de la distribución de Java como las que se crean en el desarrollo de los programas
- Las clases **heredan la implementación** de toString que tiene la clase **Object**, que es la clase que está en el nivel más alto de la jerarquía de Java

# Clases: Método toString

- La implementación que se hereda de Object es muy genérica y realmente **no aporta información sobre el contenido** del objeto, ya que solamente se compone del nombre de la clase a la que pertenece el objeto y de un código hash que lo identifica

```
public String toString()
{
    return getClass().getName() + '@' + Integer.toHexString(hashCode());
}
```

- Ejemplo:** representación textual del objeto **australia**, que es de tipo **Continente**

```
System.out.println("Australia: " +  
    australia.toString());
```



Australia: risketse.Continente@15db9742

## Clases: Método toString

- Es necesario **reimplementar** el método toString para devolver una representación en texto con aquellas características que se consideren más relevantes para describir el objeto

```
@Override
```

```
public String toString() {
    String paisesFrontera= new String();
    for(int i=0;i<frontera.size();i++)
        if(i!=frontera.size()-1)
            paisesFrontera+= "\t\t\t" + frontera.get(i).getNombre() + ",\n";
        else paisesFrontera+= "\t\t\t" + frontera.get(i).getNombre();
    String paisesContinente= new String();
    for(int i=0;i<paises.size();i++)
        if(i!=paises.size()-1)
            paisesContinente+= "\t\t\t" + paises.get(i).getNombre() + ",\n";
        else paisesContinente+= "\t\t\t" + paises.get(i).getNombre();
    String toString=
        "{" + " \n\ttnombre: " + nombre + ",\n" +
        "\tfrontera: [" + " \n" + paisesFrontera + " \n" +
        "\t\t]" + ",\n\tpaises: [" + " \n" +
        paisesContinente + " \n\t\t] \n }" + "\n";
    return toString;
}
```

```
{
  nombre: Australia,
  frontera: [
    Sudeste Asiático
  ],
  paises: [
    Indonesia,
    Nueva Guinea,
    Australia Occidental,
    Australia Oriental
  ]
}
```

# Clases: Método toString

- A partir de Java 15 se ha simplificado y mejorado el uso de las cadenas de texto multi-línea, de modo que ya no es necesario usar específicamente ni saltos de línea ni indentados

```
String toString=  
    "{" + " \n\tnombre: " + nombre + ",\n" +  
    "\tfrontera: [" + " \n" + paisesFrontera + " \n" +  
    "\t\t]" + " ,\n\tpaíses: [" + " \n" +  
    paisesContinente + " \n\t\t] \n }" + " \n";  
return toString;
```

```
String toString= ""  
{  
    nombre: %s,  
    frontera: [  
        %s  
    ],  
    países: [  
        %s  
    ]  
}"".formatted(nombre, paisesFrontera, paisesContinente);
```

Bloques de texto



```
{  
    nombre: Australia,  
    frontera: [  
        Sudeste Asiático  
    ],  
    países: [  
        Indonesia,  
        Nueva Guinea,  
        Australia Occidental,  
        Australia Oriental  
    ]  
}
```

# Clases: Método toString

- Buenas prácticas de programación (VII)

Todas las clases de un programa que necesitan mostrar sus características por consola o en una interfaz gráfica deberían de reimplementar el método toString



- Buenas prácticas de programación (VIII)

No es necesario incluir en la representación textual los valores de todos los atributos del objeto, sino solamente aquellos que se consideren relevantes o representativos





# Registros

- Un **registro** es una clase cuyas **instancias son inmutables**, es decir, no pueden cambiar los valores de los atributos en tiempo de ejecución. Tiene las siguientes restricciones:
  - Todos sus atributos son **privados**
  - Todo atributo **tiene su getter correspondiente**
  - Los atributos **no tienen los setters** correspondientes ni ningún otro método que pueda modificar los valores de los atributos
  - Tienen un constructor, denominado **constructor canónico**, con sus correspondientes argumentos para cada uno de los atributos
  - El método **toString** incluye el nombre de la clase y el nombre de los atributos con sus correspondientes valores

# Registros

```
import java.util.ArrayList;
public record Continente(ArrayList<Pais> paises,
                        String nombre,
                        String color) {
}
```



La clase Continente no tiene ningún método que permita modificar el valor de los atributos, por lo que los atributos de las instancias de la clase Continente serán inmutables, es decir, una vez toman un valor (establecido en el constructor), no podrá ser cambiado.

La gran ventaja de un registro es que para definirlo hay que escribir mucho menos que para la clase equivalente

```
public final class Continente {
    private final ArrayList<Pais> paises;
    private final String nombre;
    private final String color;

    public Continente(ArrayList<Pais> paises, String nombre, String color) {
        this.paises = paises;
        this.nombre = nombre;
        this.color = color;
    }

    public ArrayList<Pais> paises() { return paises; }

    public String nombre() { return nombre; }

    public String color() { return color; }

    @Override
    public boolean equals(Object obj) {...}

    @Override
    public int hashCode() { return Objects.hash(paises, nombre, color); }

    @Override
    public String toString() {
        return "Continente[" +
            "paises=" + paises + ", " +
            "nombre=" + nombre + ", " +
            "color=" + color + ']';
    }
}
```

# Registros

Se puede reimplementar el constructor canónico especificando todos los argumentos o usan una **declaración compacta**

Se pueden introducir constructores adicionales siempre que en su cuerpo se invoque al constructor canónico del registro

Se pueden introducir métodos funcionales siempre que no necesiten modificar los valores de los atributos

```
public record Continente(ArrayList<Pais> paises,  
                        String nombre,  
                        String color) {  
  
    // Reimplementar constructor canónico: forma compacta  
    public Continente {  
        Objects.requireNonNull(paises);  
        Objects.requireNonNull(nombre);  
        Objects.requireNonNull(color);  
    }  
  
    // Nuevo constructor (debe invocar al canónico)  
    public Continente(ArrayList<Pais> paises, String nombre) {  
        this(paises, nombre, color: "Blanco");  
    }  
  
    // Método funcional: no puede escribir sobre los atributos  
    public boolean esGrande() {  
        if(paises.size()>10) return true;  
        return false;  
    }  
}
```

# Ejecución de un programa en Java

- Todo programa de Java tiene una **clase principal** en la que se encuentra el método **main**, que será el punto a partir del cual se inicia la ejecución del programa

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

- Típicamente en el método main se crean uno o varios objetos, o instancias, que ejecutan las distintas partes del programa **a través de sus constructores** o de otros métodos
- El método **main** suele tener **muy pocas líneas de código**, ya que la codificación del programa recae en otras clases que soportan la funcionalidad requerida

# Ejecución de un programa en Java

- La clase principal **no debe tener ningún atributo**
- **Ejemplo:** juego en el que los usuarios introducen comandos para realizar las operaciones necesarias

```
public class RiskETSE {  
    public static void main(String[] args) {  
        new Menu();  
    }  
}
```

La clase **Menu** analiza los comandos que introduce el usuario y genera los objetos que invocarán los métodos con los que se da respuesta a las funcionalidades del programa