TEMA 2: CLASES Y TIPOS DE DATOS

TIPOS DE DATOS PRIMITIVOS

Data Type	Default Value (for fields)
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
boolean	false

- Los TIPOS DE DATOS PRIMITIVOS son tipos de datos **predefinidos** con las siguientes **características**:
  - Tienen una **correspondencia directa** con { los tipos de datos que es posible **representar en un computador**. / los tipos de datos de otros **lenguajes basados en procedimientos** (como C). }
  - Tienen **tamaño fijo**.
  - En el momento de declaración de una variable se realiza **automáticamente** la **reserva de memoria**.
  - **No tienen métodos.**

WRAPPERS

Primitive Data Type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

Los tdds primitivos **no son clases**, pues no encapsulan atributos ni métodos, si no que están directamente vinculados a los **valores de las variables**.
Los **wrappers** se han definido para que el esquema de los tdds de Java sea consistente con la POO.

- Los WRAPPERS son **clases de Java** que { encapsulan el **valor** del tdd. / proporcionan **métodos** que facilitan operaciones comunes sobre los datos. }
  - ↳ Estos métodos permiten { **obtener** el valor del dato. / **convertir** el dato a una **cadena** de texto y viceversa. / **convertir** el dato a otros **tdd primitivos**. }

```
Integer a= new Integer(10);
System.out.println("Valor de a en decimales: " + a.doubleValue());
Integer b= new Integer("18");
System.out.println("Valor de b: " + b);
```

⇨

```
run:
Valor de a en decimales: 10.0
Valor de b: 18
```

AUTOBOXING Y UNBOXING

El uso de wrappers proporciona a Java una gran **versatilidad** en el manejo de datos, pero genera código que es más **difícil de entender**.
El **autoboxing** y **unboxing** resuelve este problema, permitiendo elegir en cada momento si queremos tratar con tdd primitivos o wrappers.

- AUTOBOXING ⟶ convierte un **tdd primitivo** a un objeto de la correspondiente clase **wrapper**.
  - Se usa cuando { un método que tiene como argumento un wrapper recibe un valor de un tdd primitivo. / a un objeto de tipo wrapper se le asigna un valor de un tdd primitivo. }
- UNBOXING ⟶ convierte un objeto de una clase **wrapper** al **tdd primitivo** correspondiente.
  - Se usa cuando { un método que tiene como argumento un tdd primitivo recibe un objeto de un wrapper. / a un tdd primitivo se le asigna un objeto de un wrapper. }

> 👍 Normalmente, es **preferible** usar **tdd primitivos** porque simplifican el código.
> 👍 Es mejor usar **wrappers** para **convertir entre tdds**, sobre todo cuando en esa conversión se manejan **cadenas de texto**.

```
public static void main(String[] args) {
    Integer aa= 10;
    Integer bb= 20;
    if(aa>bb)
        System.out.println("aa + bb = " + suma(aa, bb));
}
public static int suma(int a, int b) {
    return a+b;
}
```
autoboxing (Integer aa= 10; Integer bb= 20;) — unboxing (suma(aa, bb))

No se puede hacer aa>bb de forma directa, pero el compilador de Java traduce en tiempo de ejecución esa condición a aa.intValue()>bb.intValue()

```
public static void main(String[] args) {
    int aa= new Integer(10);
    int bb= new Integer(20);
    if(aa>bb)
        System.out.println("aa + bb = " + suma(aa, bb));
}
public static int suma(Integer a, Integer b) {
    return a+b;
}
```
unboxing (int aa= new Integer(10); int bb= new Integer(20);) — autoboxing (suma(aa, bb))

El principal beneficio del unboxing se encuentra cuando los métodos tienen como argumentos los wrappers

REFERENCIAS

- En Java los **nombres de los objetos** son REFERENCIAS a la posición de memoria que está reservada para ellos.
  - Cuando se asigna un objeto A a otro B, **no se realiza una copia** de la memoria que ocupa A en la posición de memoria que ocupa B.
    En realidad, **se asigna la referencia** de A a la de B, es decir, ambas apuntan a la misma posición de memoria. Por tanto, **la memoria de A ya no está disponible.**

ALIASING

- El ALIASING es la asignación de referencias entre dos objetos.
- Es una de las principales características de la mayoría de los lenguajes POO, incluido Java.
- **Evita la encapsulación** de datos pues permite modificar el valor de atributos desde métodos de fuera de sus clases.
  - ↳ Por tanto, los programas son mucho más difíciles de mantener, ya que los atributos de tipo objeto se pueden modificar sin ningún tipo de control de su clase.

```
21  public static void main(String[] args) {
22      Jugador jugador= new Jugador("Luis", Valor.EJERCITO_AZUL, mapa);
23      ArrayList<Pais> paises= jugador.getPaises();
24      paises.remove(0);
```

En la línea 23 tiene lugar aliasing, ya que jugador.getPaises() devuelve el objeto paises, de tipo ArrayList<Pais>, que contiene el conjunto de países asignados al jugador
En la línea 24 se está se está eliminando el primer objeto de la lista paises, eliminado también el primer objeto del atributo de jugador que contiene la lista de países, ya que, por aliasing, apunta a la misma dirección de memoria

CÓMO Y CUÁNDO EVITAR EL ALIASING

- Es muy **difícil desarrollar** programas en los que nunca se use el aliasing
  Evitar el aliasing **reduce** enormemente el **rendimiento** } ⟶ se debe seleccionar muy bien cuándo se evitará, pues no siempre es adecuado hacerlo.
  - ↳ La asignación entre dos objetos implica una copia completa en el montón de una zona a otra.
- El aliasing sólo se puede evitar si se introduce manualmente código que genere una **nueva referencia del mismo objeto**.
  Este código tiene que { **reservar** memoria para el objeto. / **copiar** los **atributos** del objeto. }
  - ↳ Si son objetos a su vez, se debe generar una nueva referencia para ellos también.
  - – Si método **devolvía una referencia a un atributo** ⟶ ahora creará y devolverá un objeto del mismo tipo del atributo que ocupa una posición de memoria distinta.
  - – Si un método **aceptaba como entrada la referencia de un objeto** ⟶ creará un objeto del mismo tipo del argumento que ocupa una posición de memoria distinta.
- Para facilitar estas operaciones, Java dispone del método *clone*, que puede implementarse en todos los objetos.

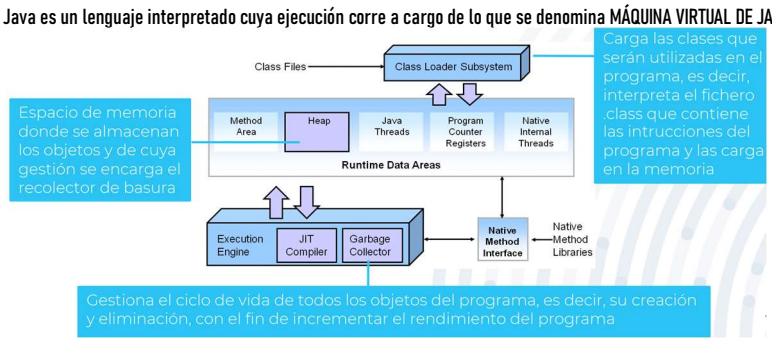
MÉTODO CLONE

- El método *clone* genera una **copia exacta** de un objeto y la almacena en una **posición diferente** de la que ocupa el objeto original.
  - El programador debe implementar el método para cada clase de cuyos objetos se desea realizar una copia, pero **no** se debe implementar para **todas** las clases de un programa, sobre todo si se pretenden realizar copias profundas.
    - COPIA PROFUNDA ⟶ se reserva memoria para todos los atributos del objeto, incluso todos los **elementos de un conjunto de datos**.

```java
@Override
public Object clone() {
    try {
        super.clone();
    } catch(CloneNotSupportedException exc) {
        System.out.println(exc.getMessage());
    }
    Jugador jugador= new Jugador(nombre, color, (Mapa) mapa.clone());
    ArrayList<Pais> paisesClonados= new ArrayList<>();
    for(int i=0;i<paises.size();i++)
        paisesClonados.add((Pais) paises.get(i).clone());
    return jugador;
```

MÁQUINA VIRTUAL DE JAVA

- Java es un lenguaje interpretado cuya ejecución corre a cargo de lo que se denomina MÁQUINA VIRTUAL DE JAVA.



ALMACENAMIENTO

ALMACENAMIENTO DE DATOS

- Dependiendo del **tipo de dato** y del **lugar** del programa en el que se definen, los datos se almacenan en zonas de memoria diferentes.

PILA (stack)

- PILA ⟶ zona de memoria a la que el procesador tiene acceso directo gracias al **puntero de pila** (sp) y en la que la lectura y escritura de datos es **rápida** y **eficiente**.
- Almacena datos relativos a la **ejecución de un método** ⟶ cuando este finaliza se **eliminan automáticamente** de memoria.
- Los datos que se almacenan en la pila siempre deben tener **tamaño conocido**:
  - ↳ El compilador necesita conocer con antelación cuánta memoria se va a reservar en la pila para mover el sp al tope.
    - El **código** correspondiente a los métodos (call stack).
    - Todos los **tdd primitivos** usados durante la ejecución de los métodos { variables locales, valores de los argumentos, valores de retorno, resultados parciales }.
    - **Las referencias a objetos creados en el programa**.

MONTÓN (heap)

- MONTÓN ⟶ zona de la memoria en la que el procesador no necesita conocer qué **cantidad de datos** se va a reservar ni cuánto **tiempo** van a estar disponibles.
- Almacena los **objetos** creados durante la ejecución de los programas.

ALMACENAMIENTO DE MÉTODOS

- Una clase no ocupa memoria ⟶ las instrucciones de los **métodos** se cargan en memoria (en la **pila**) **cuando se crea un objeto** de la clase a la que pertenecen.

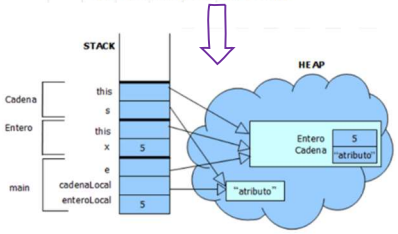
```java
public class EjemploStackYHeap {
    private int Entero;
    private String Cadena;

    public void setEntero (int x) {
        atributoEntero = x;
    }

    public void setCadena(String s) {
        Cadena = s;
    }
}

public clase Principal {
    public static void main(String[] a)
    {
        int enteroLocal = 5;
        String cadenaLocal = "atributo";

        EjemploStackAndHeap e;
        new EjemploStackAndHeap();
        e.setEntero(enteroLocal);
        e.setCadena(cadenaLocal);
```



RECOLECTOR DE BASURA

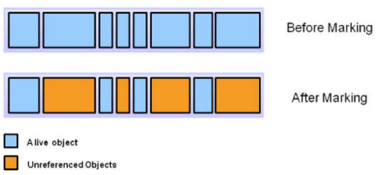
- El RECOLECTOR DE BASURA se encarga de buscar en la memoria del programa para { **identificar** cuáles objetos están en uso y cuáles no, **eliminar** los objetos que ya no se usan más (BASURA) }.
  - ↳ Este proceso se lanza durante la ejecución de un programa de Java y se ejecuta en segundo plano, realizando a su gestión de memoria.

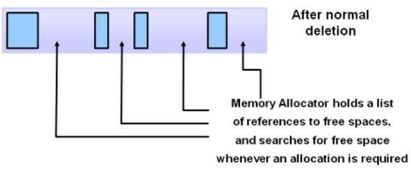
✓ Tiene un impacto directo en el **rendimiento** de un programa, ya que la eliminación de objetos que ya no se usan facilita el acceso a aquellos que sí están en uso.

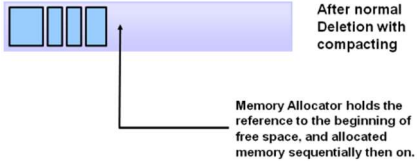
✓ Realiza una **gestión de la memoria** más **eficiente**, pues evita los errores que se cometen en la gestión manual.

PROCESO BÁSICO DE MARCADO Y BORRADO

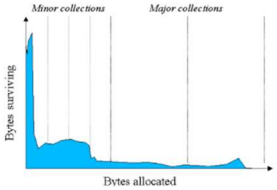
1. **Marcado** ⟶ se identifica qué zonas de memoria están siendo usadas y cuáles no.
2. OPCIÓN A: **Borrado normal** ⟶ se eliminan de memoria los objetos sin referencias asignadas durante más tiempo y se mantiene una **lista** de posibles referencias a partes de **memoria usables**.
   OPCIÓN B: **Borrado con compactación** ⟶ se eliminan de memoria los objetos sin referencias asignadas durante más tiempo y se **compacta** la memoria, moviendo los objetos restantes a posiciones de memoria consecutivas.





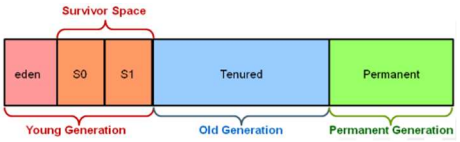


- **Problema** ⟶ las operaciones de marcado y compactación son muy **ineficientes**, ya que el recolector analiza continua y automáticamente el estado de **todos** los objetos en memoria.
- **Solución** ⟶ cambiar el esquema de la gestión de memoria.



ESTRUCTURA DEL MONTÓN

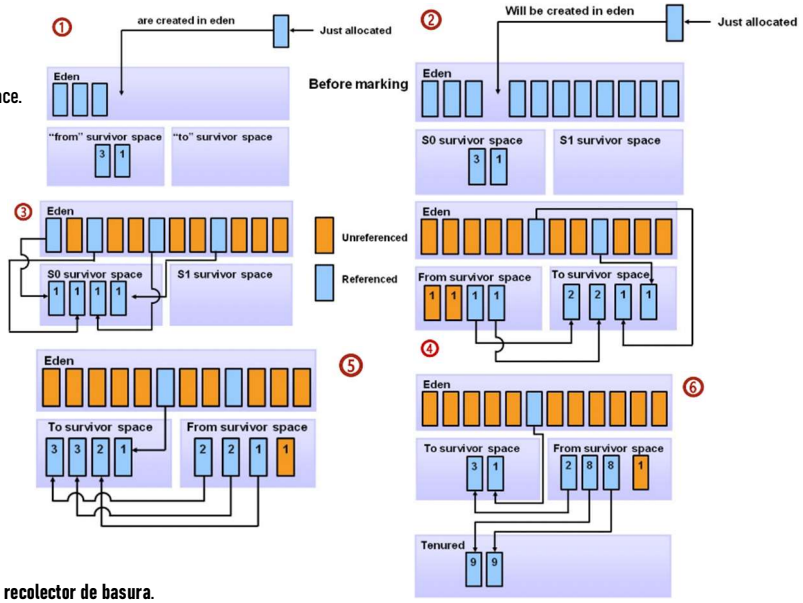
- Normalmente, el uso y eliminación de objetos **no** sigue un comportamiento **uniforme** a lo largo del tiempo,
  pues { la mayoría de objetos tienen un tiempo de supervivencia pequeño.
  { conforme pasa el tiempo se mantienen en memoria menos objetos.



- La memoria del montón se divide en GENERACIONES para facilitar la gestión de vida de los objetos.
  - YOUNG GENERATION → donde se almacenan los objetos recién creados, a los que se les asigna una fecha.
    - ↳ Está formada por el **eden space** y el **survivor space** (S0 y S1).
    - Cuando se llena se lanza una recolección de basura menor donde todos los threads se paran (**RB MINOR**).
      Es muy rápida y puede ser optimizada si se tienen que eliminar muchos objetos.
    - Los objetos que hayan sobrevivido hasta llegar a la **edad umbral** se trasladan a la generación vieja.
  - OLD GENERATION → donde se almacenan los objetos de larga duración.
    - De manera mucho menos frecuente se lanza una recolección de basura mayor donde todos los threads se paran (**RB MAYOR**).
      Es mucho más lenta que la menor porque involucra a todos los objetos vivos.
  - PERMANENT GENERATION → almacena las clases y métodos usados durante la ejecución del programa.
    - Se llena en tiempo de ejecución con metadatos (datos que describen datos) sobre las clases que se utilizan durante la ejecución del programa (**carga dinámica**).
    - En versiones posteriores se ha sustituido por el **metaspace**.

FUNCIONAMIENTO

1. Al inicio del programa la generación joven estará vacía.
2. Cualquier objeto recién creado siempre se almacenará inicialmente en el eden space.
3. Cuando el eden se llena, se lanza la primera RB minor.
   3.1. Los objetos usados del eden pasan al S0.
   3.2. Se eliminan del eden los objetos no usados.
4. En la segunda RB minor.
   4.1. Los objetos usados del eden pasan al S1.
   4.2. Los objetos usados del S0 pasan al S1 aumentando su edad.
   4.3. Se eliminan del S0 los objetos no usados.
5. En la tercera RB minor.
   5.1. Los objetos usados del eden pasan al S0.
   5.2. Los objetos usados del S1 pasan al S0 aumentando su edad.
   5.3. Se eliminan del S1 los objetos no usados.
6. Se repiten 5 y 6→ en cada RB minor se comprueba si los objetos superan una determinada edad, en cuyo caso son promocionados a la generación antigua, aumentando su edad.
7. En la generación antigua se lanzarán RB mayores con menos frecuencia.



GESTIÓN AVANZADA DE REFERENCIAS

En Java existen 4 **tipos de referencias** que se diferencian entre sí en **cómo las maneja el recolector de basura**.
↳ Son heredados de la clase $Reference$.

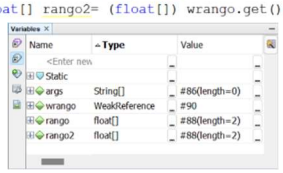
- REFERENCIAS FUERTES
  - Se **crean** automáticamente cuando se instancia un objeto → son las referencias **por defecto**.
  - El recolector de basura las **elimina** de memoria cuando apuntan a $null$.
    - ↳ Esto también le ocurre a las referencias de los objetos locales creados en un método cuando finaliza la ejecución de este.
- REFERENCIAS DÉBILES
  - Se **crean** instanciando la clase $WeakReference$ poniendo como **argumento** la referencia fuerte a la que apunta.
    - ↳ Aunque se haya creado una referencia débil, el recolector de basura puede eliminar aun así la referencia fuerte a la que apunta.
  - El **acceso** a la referencia fuerte con no está asegurado, podría devolver $null$ → se realiza con get.
  - El recolector de basura las **elimina** de memoria cuando no apuntan a una referencia fuerte o su referencia fuerte es $null$.
- REFERENCIAS SUAVES
  - Se **crean** instanciando la clase $SoftReference$ poniendo como **argumento** la referencia fuerte a la que apunta.
  - El **acceso** a la referencia fuerte está asegurado, aunque se elimine.
  - El recolector de basura las **elimina** de memoria sólo cuando sea absolutamente necesario disponer de memoria.
- REFERENCIAS FANTASMA
  - Se **crean** instanciando la clase $PhantomReference$ poniendo como **argumentos** la referencia fuerte a la que apunta y una cola donde esta se almacenará.
    - ↳ La cola es una instancia de la clase $ReferenceQueue$.
  - El **acceso** a la referencia fuerte está asegurado, aunque se elimine → se realiza a través de la cola con $poll$.
    - ↳ get siempre devuelve $null$ → las referencias fantasmas están pensadas para acceder a la memoria cuando las referencias fuertes no están disponibles.

USOS DE LOS TIPOS DE REFERENCIAS

- Referencias **débiles** → para acceder dinámicamente a la referencia fuerte de un objeto hasta que este ya no esté disponible, evitando crear referencias indiscriminadas (aliasing).
- Referencias **suaves** y **fantasmas** → para hacer **cachés en memoria**, de manera que se pueda acceder a referencias fuertes que ya no están disponibles en memoria.

```
float[] rango= new float[2];
WeakReference wrango= new WeakReference(rango);
wrango= null;
```


```
float[] rango= new float[2];
WeakReference wrango= new WeakReference(rango);
float[] rango2= (float[]) wrango.get();
```



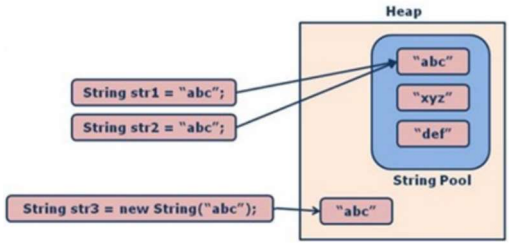

```
float[] rango= new float[2];
SoftReference srango= new SoftReference(rango);
rango= null;
float[] rango2= (float[]) srango.get();
```

- Después de eliminar la referencia fuerte de memoria de *rango* (= null), aún se mantiene el acceso a la memoria a la que apuntaba esta referencia a través de la referencia suave *srango*
- El recolector de basura únicamente eliminará el acceso a la memoria a la que apuntaba inicialmente *rango* (srango.get( ) = null) cuando no haya suficiente memoria

```
Sensor s1= new Sensor(50, new float[]{ -10, 10 });
// Referencia fantasma
ReferenceQueue queue= new ReferenceQueue();
PhantomReference phSensor= new PhantomReference(s1.getRango(), queue);
Reference<Sensor> sensorRef= queue.poll();
// Acceso a las referencias
System.out.println("Acceso a la referencia: " + sensorRef.get());
if(sensorRef!=null)
    System.out.println("Referencia de s1: " + sensorRef.get());
```

CLASE STRING

- Los objetos que son cadenas de texto se pueden **crear** de dos formas diferentes:
  - **Directamente** ⟶ se asigna una cadena de texto al objeto.
    - La cadena de texto se almacena en una zona del montón llamada STRING POOL para que cada vez que se asigne la misma cadena a otra referencia, esta apunte al mismo string en el String pool.
  - **Indirectamente** ⟶ el objeto se crea usando un constructor de String.
    - La cadena de texto se almacena en el montón fuera del Sring pool, aunque tenga el mismo valor que alguna cadena previa.



Si se van a realizar múltiples **modificaciones** sobre una cadena de texto, no se debería usar la clase $String$.
En su lugar se debería usar la clase $StringBuffer$ ya que **no reserva** espacio de memoria **cada vez que se modifica** una cadena de texto.

- En realidad, una cadena de texto es un objeto **inmutable** independientemente de cómo se cree ⟶ una vez se ha reservado memoria y asignado un valor, este no se puede modificar.
  - Cuando usamos un método que parece modificar una cadena de texto, en realidad está creando una nueva cadena de texto con el valor modificado en alguna parte de la memoria.
  - Entonces, aunque $String$ sea una clase de Java, no se comporta como el resto respecto a la **asignación entre objetos**.
  - Como consecuencia, el uso indiscriminado de cadenas de texto puede llevar a una merma importante en el **rendimiento** de un programa.

```
String australia= "australia";
String continente= australia;
String nuevoContinente= continente;
```

Se crea un único objeto (asignación directa), mientras que en los otros dos casos se sigue la regla de las referencias entre objetos

IDENTIDAD DE OBJETOS – MÉTODO EQUALS

¿Cuándo se puede considerar que dos objetos son iguales?

A. Dos objetos son iguales si ocupan la **misma posición de memoria**.
   - Condición **muy restrictiva**, ya que podrían existir dos objetos con los mismos atributos en posiciones de memoria distintas.

B. Dos objetos son iguales si son del **mismo tipo** y si los valores de todos sus **atributos** también son **iguales**.
   - Obliga a que los dos objetos tengan los mismos valores de los atributos en el momento de la comparación y que los atributos sean inmutables.

- $equals$ es un método que indica si un objeto (el argumento) es igual a otro (desde el que se invoca).
  - Todas las clases lo tienen, pues es heredado de la clase $Object$.
  - La implementación heredada sigue la **opción 1**, ya que lo único que se conoce de cualquier objeto es su referencia.
  - Se debe **reimplementar** comparando los atributos inmutables, de modo que dos objetos son iguales si estos atributos tienen el mismo valor.

```
Continente australia1= new Continente("Australia", "AZUL");
Continente australia2= new Continente("Australia", "AZUL");
```

Aunque ocupan posiciones de memoria diferentes, parece razonable pensar que los objetos **australia1** y **australia2** representan al mismo continente, es decir, los dos objetos son iguales

```
Jugador jugador1= new Jugador("Marta", "AZUL", 14);
Jugador jugador2= new Jugador("Marta", "AZUL", 14);
jugador2.setNumEjercitos(20);
```

El número de ejércitos es un **atributo mutable** de la clase **Jugador**, que depende de los resultados del juego, de modo que con la opción 2, **jugador1** sería igual a **jugador2** solamente cuando el valor de ese atributo coincida

```
@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (obj == null) {
        return false;
    }
    if (getClass() != obj.getClass()) {
        return false;
    }
    final Jugador other = (Jugador) obj;
    if (!this.nombre.equals(other.nombre)) {
        return false;
    }
    if (!this.color.equals(other.color)) {
        return false;
    }
    return true;
}
```

Si la referencia de los dos objetos es la misma, entonces los objetos son iguales

Si el objeto con el que se compara es **null**, entonces los objetos **no** son iguales

Si la clase de los dos objetos es diferente, entonces los objetos **no** son iguales

Si los **nombres o los colores** de los dos objetos son diferentes, entonces los objetos son diferentes (se ha comprobado antes que los dos objetos pertenecen a la misma clase y que no son nulos)

EN EL EXAMEN, SIEMPRE REIMPLEMENTAR $equals$
También lo usa $contains$, los $hasmaps$, etc...

Todos los objetos que realicen comparaciones deben tener reimplementado el método $equals$.

Los atributos **inmutables** pueden ser { tdds primitivos, wrappers, Strings }.