



2.

# Clases y tipos de datos

Tipos primitivos, referencias y aliasing

# Tipos de datos primitivos

- Los tipos de datos primitivos son tipos de datos **predefinidos** tienen una correspondencia directa con los tipos de datos de otros lenguajes basados en procedimientos (como C)
  - Tienen un tamaño **fijo**
  - Tienen una **correspondencia directa** con los tipos de datos que es posible representar en un computador
  - En el momento de su declaración se realiza automáticamente la **reserva de memoria** para la variable
  - No tienen métodos

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

# Tipos de datos primitivos: Wrappers

- Los tipos de datos primitivos no son clases, puesto que no encapsulan ni datos ni métodos que acceden y modifican dichos datos, sino que están directamente vinculados a los **valores de las variables**
- Para hacer consistente el esquema de tipos de datos de Java con la programación orientada a objetos, para cada tipo de datos primitivos se han definido **wrappers** que envuelven los valores de los tipos en diferentes tipos de objetos en función de los datos primitivos

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

# Tipos de datos primitivos: Wrappers

- Estos wrappers son **clases de Java** que encapsulan el valor del tipo de dato y proporcionan una serie de métodos que facilitan operaciones comunes sobre los datos
- Los métodos de los wrappers de los tipos primitivos permiten **obtener** el valor del dato; **convertir** el dato a una cadena de texto, y viceversa; o **convertir** el dato a otros tipos de datos 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
```

# Wrappers: Autoboxing & Unboxing

- El uso de wrappers proporciona una gran versatilidad en el manejo de datos al lenguaje Java, pero genera código que es mucho **más difícil de entender** que el que genera el uso de datos primitivos

```
int a= 10;  
int b= 20;  
if(a>b)  
    System.out.println("a es mayor que b");
```

vs

```
Integer aa= new Integer(10);  
Integer bb= new Integer(20);  
if(aa.intValue()>bb.intValue())  
    System.out.println("aa es mayor que bb");
```

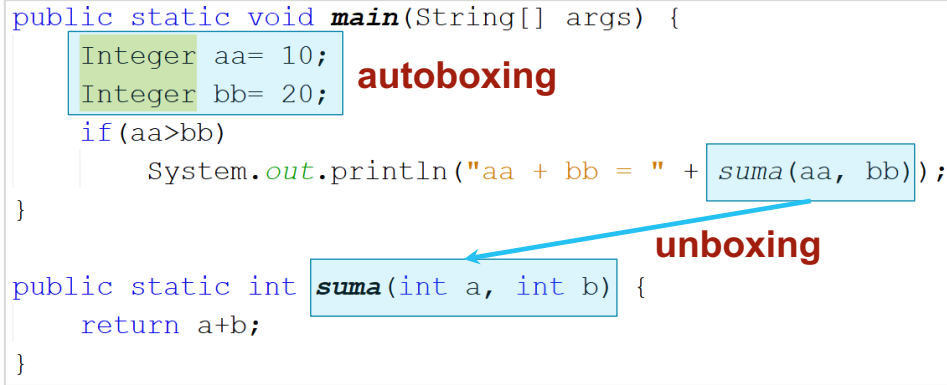
- El uso de **autoboxing** y **unboxing** resuelve este problema, ya que permite tratar un wrapper como si fuese un tipo de dato primitivo, y viceversa; pudiendo elegir en cada momento el comportamiento que se desee

# Wrappers: Autoboxing & Unboxing

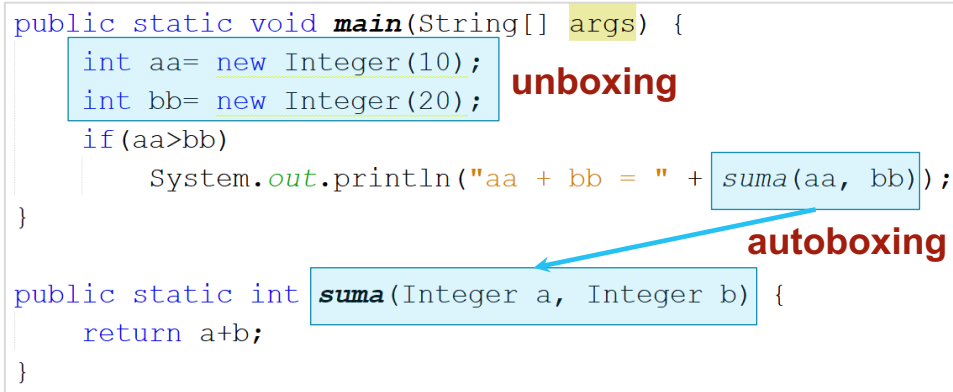
- **Autoboxing**: convierte un tipo de dato primitivo a un objeto de la correspondiente clase wrapper
  - Se aplica autoboxing cuando **(a)** un método que tiene como argumento un tipo wrapper recibe un valor de un tipo primitivo; o **(b)** a un objeto de tipo wrapper se le asigna un valor de un tipo primitivo
- **Unboxing**: convierte un objeto de una clase wrapper al tipo de dato primitivo correspondiente
  - Se aplica unboxing cuando **(a)** un método que tiene como argumento un tipo primitivo recibe un objeto de una clase wrapper; o **(b)** a un tipo de dato primitivo se le asigna un objeto de tipo wrapper

# Wrappers: Autoboxing & Unboxing

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



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



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()`

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

# Wrappers: Autoboxing & Unboxing

- Buenas prácticas de programación (VIII)

En general, es preferible usar tipos de datos primitivos porque simplifican el código y hacen más sencilla su comprensión



- Buenas prácticas de programación (IX)

Es aconsejable usar wrappers a tipos de datos primitivos cuando sea necesario convertir entre tipos de datos, sobre todo cuando en esa conversión se manejan cadenas de texto





# Wrappers: Autoboxing & Unboxing

Dado el siguiente trozo de código, ¿cuáles de las siguientes afirmaciones son ciertas?

```
1  ArrayList<Integer> nums= ArrayList<>() ;  
2  Integer x= new Integer(10) ;  
3  nums.add(10) ;  
4  nums.add(x) ;  
5  Integer y= nums.get(0) ;  
6  Integer z= nums.get(1) ;  
7  z= y;  
8  y= y+10;  
9  int res= y+20;  
10 for(Integer val : nums) res+= val;
```

- ☐ En la línea 8 del código se está usando autoboxing.
- ☐ Después de que se haya ejecutado la línea 6 del código, la dirección de memoria de y es igual que la de z.
- ☐ Después de que se haya ejecutado la línea 8 del código, el valor de z es 20.
- ☐ Después de que se haya ejecutado la línea 7 del código, la dirección de memoria de z es igual que la de y.
- ☐ Se generará un error al ejecutar la línea 10.
- ☐ En la línea 9 del código se está usando unboxing.

# Referencias

- En Java los **nombres de los objetos son referencias** a la posición de memoria que está reservada para dichos objetos
- Este uso de referencias tiene importantes **implicaciones**
  - Cuando se asigna un objeto  $\text{obj}_A$  a otro objeto  $\text{obj}_B$ , en realidad **no se realiza una copia** de la memoria que ocupa  $\text{obj}_B$  en la posición de memoria que ocupa  $\text{obj}_A$ 
    - En esta situación se está realizando una **asignación de la referencia del objeto  $\text{obj}_B$  a la referencia al  $\text{obj}_B$** , o lo que es lo mismo, ambas referencias apuntan a la misma posición de memoria
    - La memoria del objeto  $\text{obj}_A$  ya **no está disponible**

# Referencias

```
public class RiskETSE {  
    public static void main(String[] args) {  
        Continente australia= new Continente("Australia", "Azul");  
        Continente asia= new Continente("Asia", "Cyan");  
        australia= asia;  
    }  
}
```

Name	Type	Value
<Enter new v		
Static		
args	String[]	#86(length=0)
asia	Continente	#91
color	String	"Cyan"
frontera	ArrayList	"size = 0"
nombre	String	"Asia"
países	ArrayList	"size = 0"
australia	Continente	#90
color	String	"Azul"
frontera	ArrayList	"size = 0"
nombre	String	"Australia"
países	ArrayList	"size = 0"

Pos. #91

Pos. #90

```
public class RiskETSE {  
    public static void main(String[] args) {  
        Continente australia= new Continente("Australia", "Azul");  
        Continente asia= new Continente("Asia", "Cyan");  
        australia= asia;  
    }  
}
```

Name	Type	Value
<Enter new v		
Static		
args	String[]	#86(length=0)
asia	Continente	#91
color	String	"Cyan"
frontera	ArrayList	"size = 0"
nombre	String	"Asia"
países	ArrayList	"size = 0"
australia	Continente	#91
color	String	"Cyan"
frontera	ArrayList	"size = 0"
nombre	String	"Asia"
países	ArrayList	"size = 0"

Pos. #91

Pos. #91

Después de realizar la asignación entre las referencias australia y asia de la clase Continente (**australia = asia**), la dirección de memoria a la que apuntan es exactamente la misma (#91)

# Referencias

- La asignación de referencias entre dos objetos se denomina **aliasing** y es una de las principales características no solo de Java, sino de la mayoría de los lenguajes orientados a objetos
- El uso de aliasing **evita la encapsulación de los datos**, ya que permite modificar el valor de los atributos desde métodos que no pertenecen a las clases que contienen dichos atributos

The big lie of object-oriented programming is that objects provide encapsulation

*John Hogg. Islands: Aliasing protection in object-oriented languages. Proceedings OOPSLA'91. SIGPLAN Notices, 26(11):271-285, 1991*

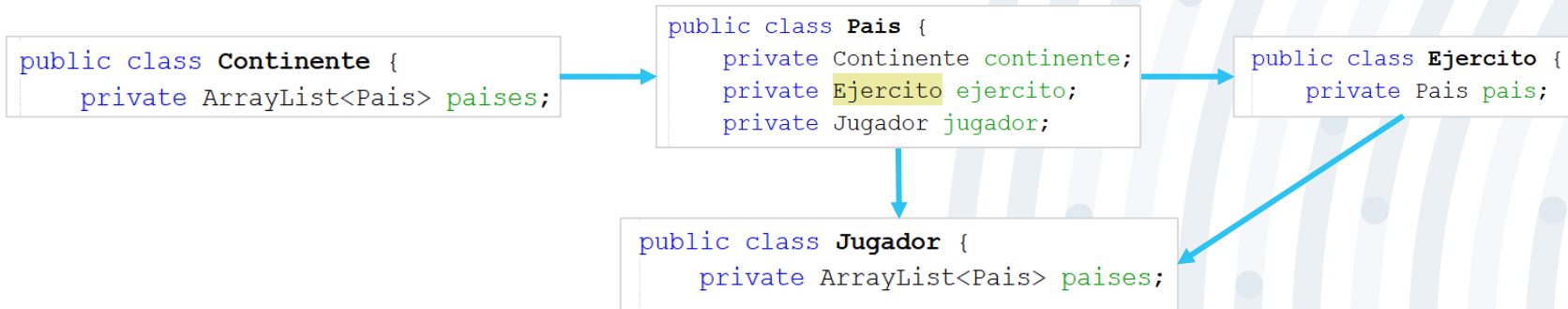
# Referencias

```
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);  
25  }
```

- Al crear el objeto **jugador** se le asignan automáticamente una serie de países y el ejército de color azul
- 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 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

# Referencias

- El aliasing tiene otros efectos no deseables
  - Los programas son **mucho más difíciles de mantener** porque los valores de los atributos de tipo objeto se podrían modificar en cualquier parte del programa sin ningún tipo de control por parte de la clase a la que pertenecen
  - Ejemplo: los métodos de Pais, Ejercito y Jugador podrían cambiar los atributos de los países de la clase Continente



# Referencias

- Evitar aliasing en los lenguajes de programación orientada a objetos reduciría enormemente su rendimiento, puesto que supondría que la asignación entre dos objetos sería una **copia completa** de una zona de memoria a otra zona de memoria del montón
- Desde un punto de vista práctico, **es muy difícil** el desarrollo de programas en los que se no se haga uso de aliasing en alguna parte del código
  - No siempre es adecuado evitar aliasing
  - Es necesario **seleccionar** aquellas parte del código en las que se debe evitar aliasing

# Referencias

- La única forma de evitar aliasing es introducir **manualmente** código que genere una nueva referencia del objeto, es decir, reserve memoria, y copie los atributos del objeto generando con nuevas referencias cuando sea necesario
  - En vez de que un método devuelva una referencia a un atributo, deberá **crear y devolver un objeto del mismo tipo** del atributo que ocupa una posición de memoria diferente
  - En vez de que un método acepte como entrada la referencia de un objeto, deberá **crear un objeto del mismo tipo** del argumento para que ocupe una posición de memoria diferente
  - Para facilitar estas operaciones Java proporciona el método **clone**, que puede implementarse en todos los objetos



# Referencias

- El método **clone** está pensado para **generar una copia exacta** de un objeto, almacenando esa copia en una posición de memoria **diferente** de la que ocupa dicho objeto
  - El programador debe implementar **explícitamente** el método clone **para cada clase** de cuyos objetos se desea realizar una copia
  - No es práctico implementar el método clone para **todas** las clases de un programa, sobre todo si se trata de realizar **copias profundas**
    - En una copia profunda es necesario reservar memoria para todos los atributos de la clase, incluyendo **todos los elementos de un conjunto de datos**

# Referencias

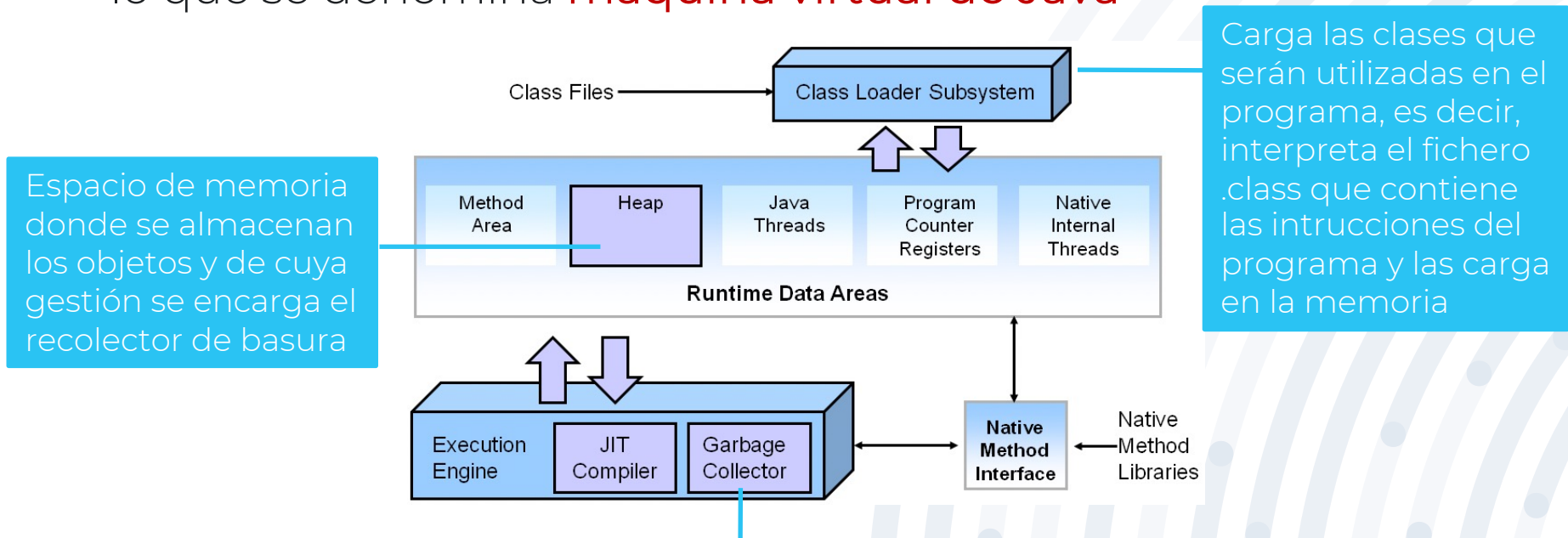
```
@Override
public Object clone() {
    try {
        super.clone();
    } catch (CloneNotSupportedException exc) {
        System.out.println(exc.getMessage());
    }
    Jugador jugador= new Jugador(nombre, color, (Mapa) mapa.clone());
    ArrayList<Pais> paísesClonados= new ArrayList<>();
    for(int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

Es necesario generar un nuevo objeto del tipo Mapa llamando al método clone, que debería estar implementado en la clase Mapa

No solamente es necesario reservar memoria para una nueva lista de países (paísesClonados), sino que para cada país se debe reservar memoria y copiar todos los valores de sus atributos a través de clone necesario

# Máquina virtual de Java

- Java es un lenguaje interpretado cuya ejecución corre a cargo de lo que se denomina **máquina virtual de Java**



Gestiona el ciclo de vida de todos los objetos del programa, es decir, su creación y eliminación, con el fin de incrementar el rendimiento del programa

# Almacenamiento: 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), zona de la memoria a la que el procesador tiene acceso directo a través de un puntero de pila, y en la que la lectura y escritura de los datos **es rápida y eficiente**
  - El compilador **debe conocer con antelación** cuánta memoria se necesita reservar en la pila, ya que debe de mover el puntero a lo largo de la pila para acceder a los datos
  - Las variables existen en la pila **durante la ejecución del método que las ha creado**, de modo que cuando finaliza su ejecución, se eliminan automáticamente de la memoria

# Almacenamiento: Datos

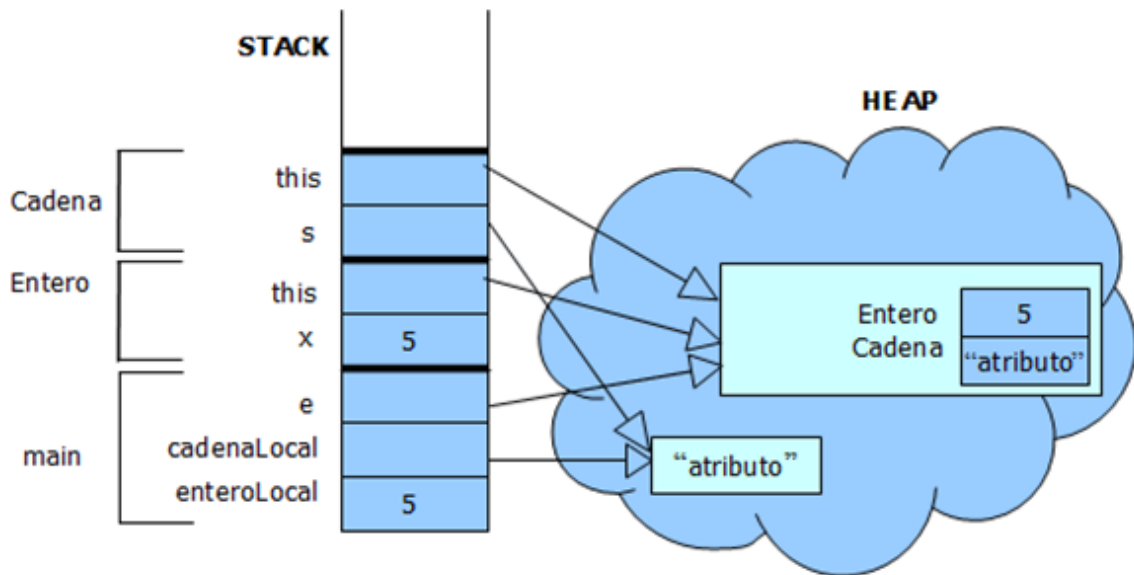
- Los datos que se almacenan en la pila siempre deben de tener un **tamaño conocido**
  - Todo el **código** correspondiente a los métodos (**call stack**)
  - Todos los **datos de tipo primitivos** usados durante la ejecución de los métodos
    - Variables locales de los métodos
    - Valores de los argumentos
    - Valores de retorno de los métodos
    - Resultados parciales que se obtienen durante su ejecución
  - Las **referencias a los objetos** creados en el programa

# Almacenamiento: Datos

- **Montón** (Heap), zona de la memoria en la que el procesador no necesita conocer qué cantidad de datos se deben reservar y cuánto tiempo van a estar disponibles esos datos
  - **Almacena los objetos** creados durante la ejecución del programa
  - La gestión del montón corre a cargo del **recolector de basura**, ya que los datos no eliminan automáticamente cuando dejan de ser necesarios
  - El **rendimiento de un programa** en Java está muy condicionado por la gestión eficiente del montón, o lo que es lo mismo, por el rendimiento del recolector de basura

# Almacenamiento: Datos

```
public class EjemploStackYHeap {  
    private int Entero;  
    private String Cadena;  
  
    public void setEntero (int x) {  
        atributoEntero = x;  
    }  
  
    public void setCadena (String s) {  
        Cadena = s;  
    }  
}  
  
public class Principal {  
    public static void main (String[] a)  
    {  
        int enteroLocal = 5;  
        String cadenaLocal = "atributo";  
  
        EjemploStackAndHeap e;  
        new EjemploStackAndHeap ();  
        e.setEntero (enteroLocal) ;  
        e.setCadena (cadenaLocal) ;  
    }  
}
```



# Almacenamiento: 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 dichos métodos

```
Continente australia= new Continente("Australia", Valor.PAIS_AZUL);
```










- A partir del momento en el que se crea el objeto, o instancia, es posible acceder a todos los métodos a través del **operador "."**

```
ArrayList<Pais> frontera= australia.obtenerFrontera();
```

- Cuando en el método se hace referencia a un atributo de la clase a la que pertenece, este atributo **tomará los valores asociados al objeto** que hace uso del operador "."



# Almacenamiento: Métodos

[-]  australia	Continente	#98
 color	String	"\u001b[44m"
 frontera	ArrayList	"size = 0"
 nombre	String	"Australia"
 pais	ArrayList	"size = 4"
 [0]	Pais	#135
 [1]	Pais	#136
 [2]	Pais	#137
 [3]	Pais	#138

**MONTÓN**

australia.paises

```
ArrayList<Pais> frontera= australia.obtenerFrontera();
```

**PILA**

```
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;  
}
```

# Recolector de basura

- El **recolector de basura** se encarga de buscar en la memoria del programa para **(a)** identificar qué objetos se encuentran en uso y cuáles no y **(b)** eliminar los objetos que ya no se usan más
  - Es un proceso que se lanza durante la ejecución de un programa de Java y que corre en segundo plano realizando automáticamente la gestión de la memoria del programa
  - Tiene un **impacto directo en el rendimiento de un programa**, puesto que la eliminación de los objetos que no se usan (**basura**) facilita el acceso a los objetos que se están utilizando en el programa
  - Realiza una gestión **más eficiente de la memoria**, evitando los errores que se cometen en la gestión manual, como ocurre en otros lenguajes como C (función **free**) o C++ (operador **delete**)

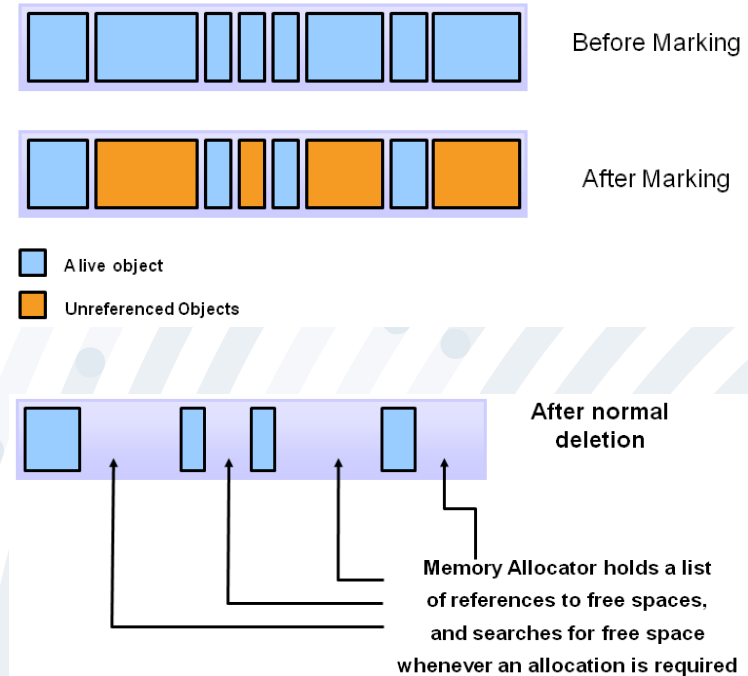
# Recolector de basura: proceso básico

- Paso 1: Marcado

Se identifican qué zonas de la memoria están siendo usadas y cuáles no, lo cual puede ser **muy ineficiente** si se deben analizar todos los objetos del sistema

- Paso 2 (opción 1): Borrado normal

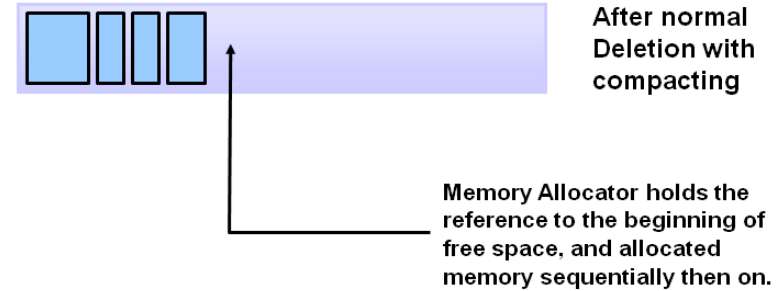
Se eliminan de memoria los objetos que no tienen referencias asignadas durante más tiempo y mantiene una lista de posibles referencias a la parte de la memoria que puede ser utilizada



# Recolector de basura: proceso básico

- Paso 2 (opción 2): Borrado con compactación

Para mejorar el rendimiento, además de borrar los objetos no referenciados, se compacta la memoria, moviendo los objetos referenciados a posiciones de memoria consecutivas



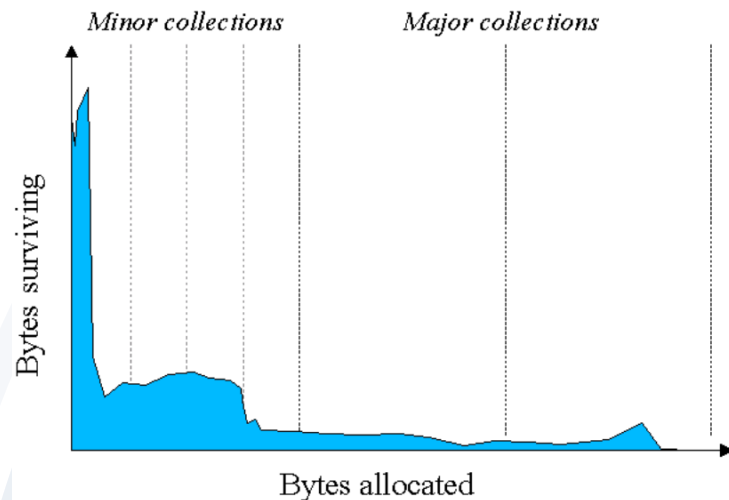
## Problema con el algoritmo de "marca y barrido"

Las operaciones de marcado y compactación **son muy ineficientes**, ya que el recolector de basura analiza continua y automáticamente el estado de los objetos en la memoria del programa ⇒ (solución) cambiar el esquema de gestión de la memoria

# Recolector de basura: Estructura

En la mayoría de los programas, el uso y la eliminación de los objetos **no sigue un comportamiento uniforme** a lo largo del tiempo

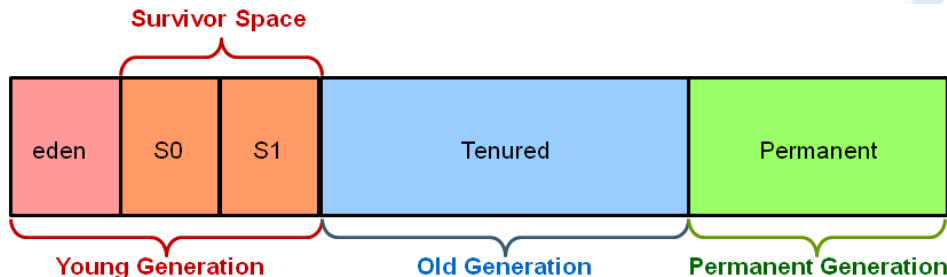
- El tiempo de supervivencia de los objetos es pequeño
- A medida que pasa el tiempo, cada vez se mantienen en memoria menos objetos



## Solución

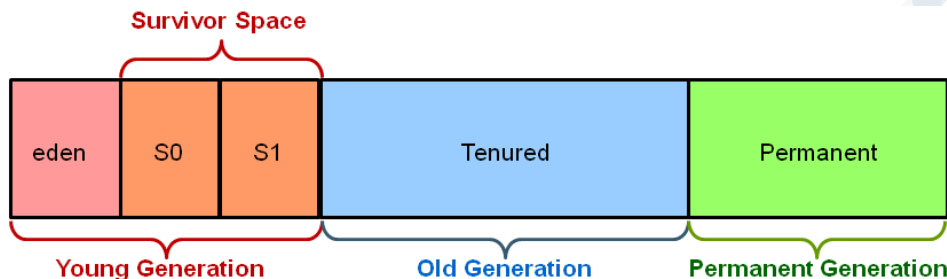
Dividir la memoria en varias partes, llamadas **generaciones**, para facilitar la gestión de la vida de los objetos

# Recolector de basura: Estructura



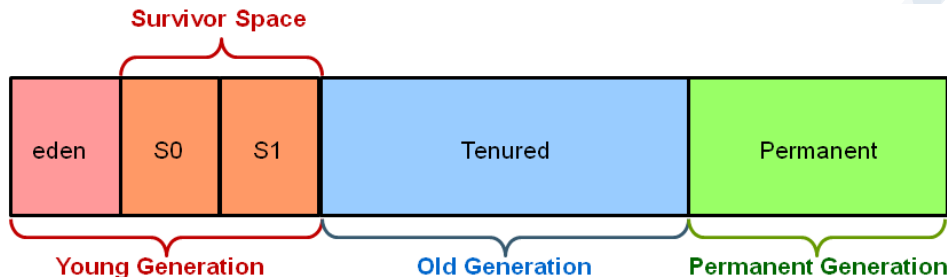
- **Young generation:** se almacenan y se les asigna una fecha a todos los objetos recién creados
  - Cuando se llena, se lanza una recolección de basura menor (**RB minor**, donde todos los threads se paran), que es muy rápida y que puede ser optimizada si se tienen que eliminar muchos objetos
  - Los objetos que no se eliminan, envejecen y pasan a la generación antigua (**old generation**)

# Recolector de basura: Estructura

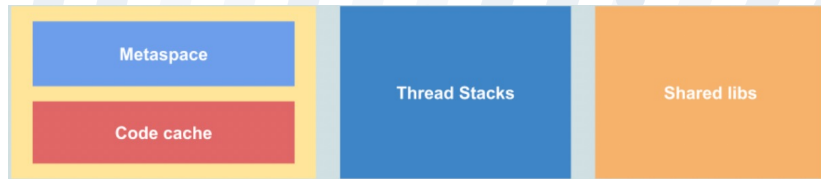


- **Old Generation:** se utiliza para almacenar los objetos de larga duración
  - Se establece un **umbral** en la región joven: los objetos con una edad mayor que ese umbral se trasladan a la generación antigua
  - Cuando se llena, se lanza una recolección de basura mayor (**RB mayor**, donde también se detienen todos los threads), que es mucho más lenta que la menor, ya que involucra a todos los objetos vivos

# Recolector de basura: Estructura

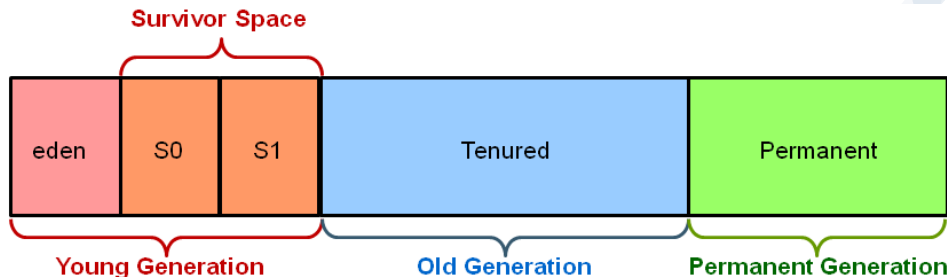


- **Permanent Generation:** se utiliza para almacenar las clases y los métodos usados durante la ejecución del programa
  - Se llena en tiempo de ejecución con metadatos sobre las clases que se utilizan durante la ejecución del programa (**carga dinámica**)
  - En versiones posteriores a Java 8 esta generación se ha sustituido por **metaspace**



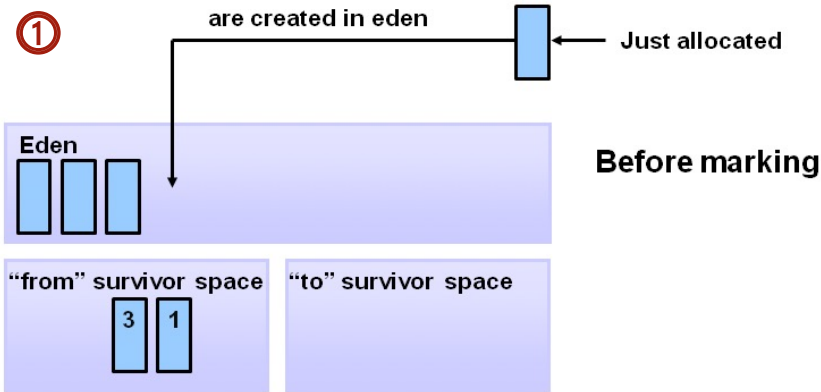


# Recolector de basura: Estructura



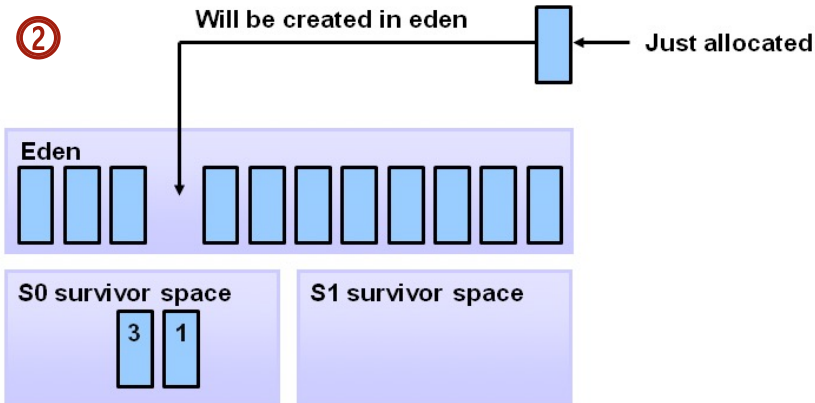
- **Permanent Generation:** se utiliza para almacenar los objetos de larga duración
  - Se establece un **umbral** en la región joven: los objetos con una edad mayor que ese umbral se trasladan a la generación antigua
  - Cuando se llena, se lanza una recolección de basura mayor (**major**, donde también se detienen todos los threads), que será mucho más lenta que la menor, ya que involucra a todos los objetos vivos

# Recolector de basura: Funcionamiento



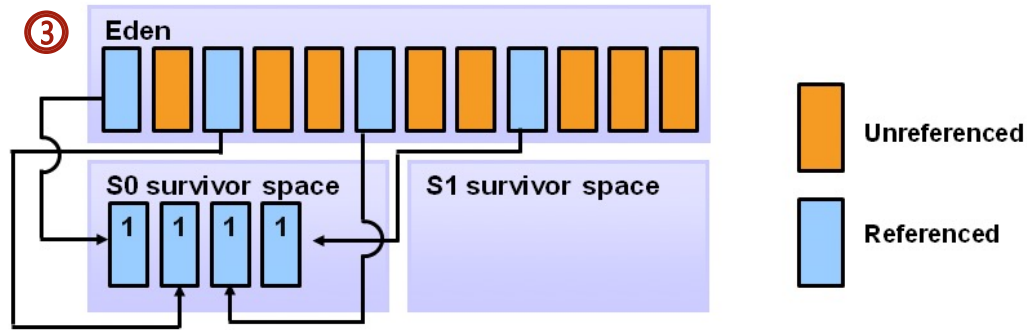
Al inicio del programa la **generación joven** está vacía, incluyendo todas partes en las cuales se estructura: **eden**, **superviviente 0** (S0) y **superviviente 1** (S1)

Cualquier **objeto recién** creado siempre se almacena inicialmente en el **eden**

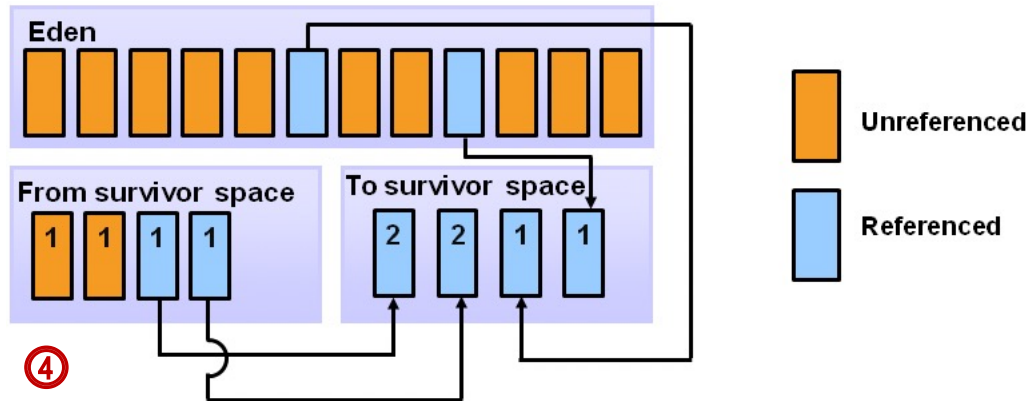


Cuando el **eden** se llena, se lanza una **recolección de basura menor** (RB menor) en la que se eliminan los objetos que no se van a usar (con referencia **null**)

# Recolector de basura: Funcionamiento

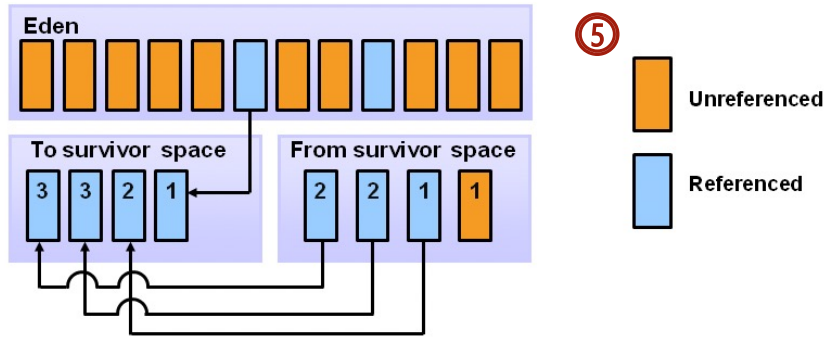


En la RB minor, **todos** los objetos que son usados (referenciados) se mueven al espacio **S0**, mientras que los objetos no usados (no referenciados) se eliminan del espacio **eden**

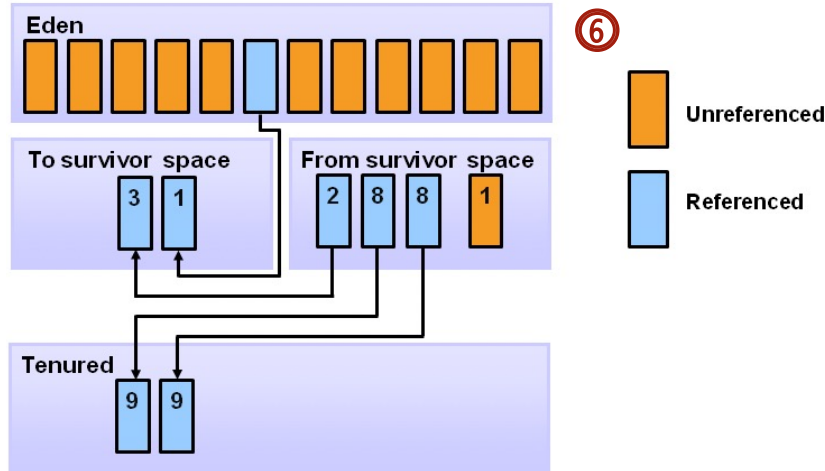


En la siguiente RB minor, los objetos usados del **eden** se mueven al espacio **S1**, los objetos usados del espacio **S0** se mueven al espacio **S1** aumentando su edad, y se eliminan del **espacio S0** los objetos no usados

# Recolector de basura: Funcionamiento

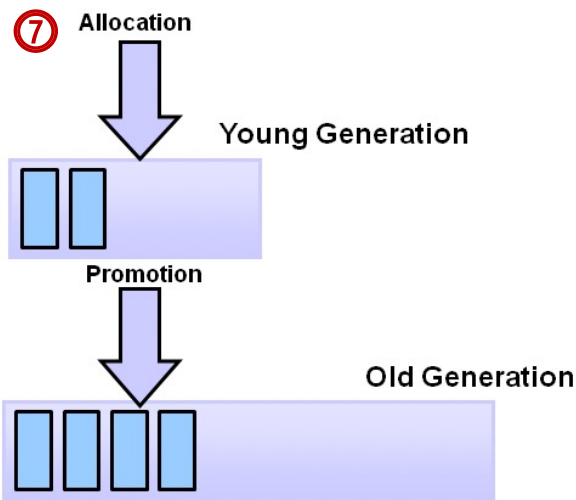


En la siguiente RB menor, **todos** los objetos usados del **eden** se mueven al espacio **S0**, mientras que los usados del espacio **S1** se mueven al espacio **S0** aumentando su edad, y se eliminan del espacio **S1** los objetos no usados



En cada RB mayor se comprueba si los objetos usados superan una determinada edad (8 en el ejemplo), en cuyo caso **son promocionados** (es decir, movidos) a la generación antigua, aumentando su edad

# Recolector de basura: Funcionamiento



**Resumen:** en la generación joven se aplican sucesivas RB minor para manejar de forma más eficiente el ciclo de vida de los objetos, asumiendo que los objetos recientes son los más usados

Los objetos de la generación antigua también pueden dejar de ser usados, por lo tanto, para eliminar los objetos no usados, en la **generación antigua se lanza un recolector de basura mayor**

Este funcionamiento del recolector de basura es genérico, existiendo **varias formas de implementarlo:**

- serial GC, parallel GC, CMS GC, G1 GC, Epsilon GC (Java 11), Shenandoah (Java 12), ZGC (Java 12)

# Referencias: Gestión avanzada

- ¿Existe alguna forma de acceder a la posición de memoria de un objeto sobre el que se ha hecho aliasing?
  - ¿Existe alguna forma acceder a la posición de memoria de un objeto una vez la referencia a esa posición no esté disponible?

```
float[] rango= new float[2];
```

```
rango[0]= 0;
```

```
rango[1]= 10;
```

```
Sensor s1= new Sensor(50, rango);
```


```
Sensor s2= new Sensor(40, rango);
```

```
// Aliasing
```

```
// Se pierde el acceso a la memoria con id= 40, rango= [0, 10]
```

```
s1= s2;
```

Después de hacer `s1= s2`, ¿sería posible acceder a la memoria que se encuentra en la posición #91



Name	Type	Value
<Enter new variable>		
Static		
args	String[]	#86(length=0)
rango	float[]	#88(length=2)
s1	Sensor	#91
rango	float[]	#88(length=2)
historial	float[]	#94(length=10)
id	int	50
s2	Sensor	#92
rango	float[]	#88(length=2)
historial	float[]	#93(length=10)
id	int	40


# Referencias: Gestión avanzada

- En Java existen cuatro tipos de referencias que se diferencian entre sí en el manejo que hace de ellas el **recolector de basura**
- Todos los tipos de referencias heredan de la clase **Reference**
  - **Referencias fuertes** (Strong references)
    - Son las referencias por defecto
    - Se crean automáticamente cuando se **instancia** un objeto
    - El recolector de basura no las elimina de la memoria hasta que apuntan a **null**, lo cual también puede ocurrir cuando se crea un objeto local en un método (**obj<sub>M</sub> != null**) y se finaliza la ejecución de dicho método (**obj<sub>M</sub> = null**)

# Referencias: Gestión avanzada

- Todos los tipos de referencias heredan de la clase **Reference**
  - **Referencias débiles** (Weak references)
    - Para crear este tipo de referencias se debe instanciar la clase **WeakReference**, teniendo como argumento una referencia fuerte **a la que apunta**

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



Name	Type	Value
<Enter new w...		
Static		
args	String[]	#86(length=0)
rango	float[]	#88(length=2)
[0]	float	0.0
[1]	float	0.0
wrango		null

- La creación de una referencia débil no obliga al recolector de basura a mantener la referencia fuerte a la que apunta




# Referencias: Gestión avanzada

- Todos los tipos de referencias heredan de la clase **Reference**
  - **Referencias débiles** (Weak references)

```
float[] rango= new float[2];
```

```
WeakReference wrango= new WeakReference(rango);
```

```
float[] rango2= (float[]) wrango.get();
```



Name	Type	Value
<Enter new		
Static		
args	String[]	#86(length=0)
wrango	WeakReference	#90
rango	float[]	#88(length=2)
rango2	float[]	#88(length=2)

- Para acceder a la referencia fuerte se puede usar el método **get**, pero no está asegurado que devuelva siempre una referencia fuerte no nula
- El recolector de basura elimina las referencias débiles cuando no apuntan a una referencia fuerte o la referencia fuerte es **null**

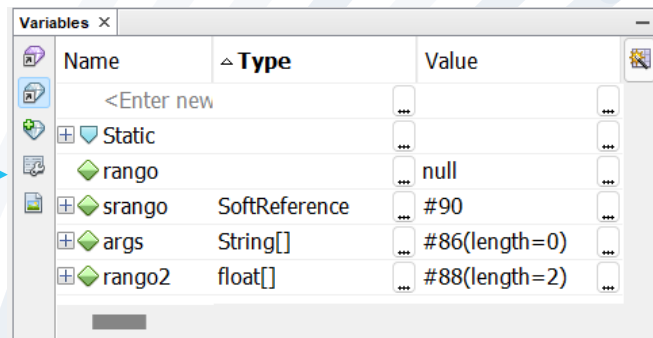
# Referencias: Gestión avanzada

- Todos los tipos de referencias heredan de la clase **Reference**
  - **Referencias suaves** (Soft references)
    - Para crear este tipo de referencias se debe instanciar la clase **SoftReference**, teniendo como argumento una referencia fuerte a la que apunta
    - Aunque la referencia fuerte a la que apunta sea eliminada por el recolector de basura (= **null**), se puede seguir accediendo a la posición de memoria apuntada inicialmente por ella
    - El recolector de basura elimina este tipo de referencias **cuando sea absolutamente necesario disponer de memoria**, manteniéndolas siempre que haya memoria abundante

# Referencias: Gestión avanzada

- Todos los tipos de referencias heredan de la clase `Reference`
  - **Referencias suaves** (Soft references)

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



Name	Type	Value
<Enter new>		
Static		
rango		null
srango	SoftReference	#90
args	String[]	#86(length=0)
rango2	float[]	#88(length=2)

- ❑ 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

# Referencias: Gestión avanzada

- Todos los tipos de referencias heredan de la clase **Reference**
  - **Referencias fantasmas** (Phantom references)
    - Para crear este tipo de referencias se debe instanciar la clase **PhantomReference**, teniendo como argumentos **(a)** una referencia fuerte a la que apunta y **(b)** una cola en la que se almacenará dicha referencia fuerte
    - Aunque la referencia fuerte a la que apunta sea eliminada por el recolector de basura (= **null**), se puede seguir accediendo a la posición de memoria apuntada inicialmente por ella ya que se almacena en la cola que se ha introducido como argumento
    - La cola es una instancia de la clase **ReferenceQueue**

# Referencias: Gestión avanzada

- Todos los tipos de referencias heredan de la clase `Reference`
  - **Referencias fantasmas** (Phantom references)
    - El método `get()` devolverá siempre `null`, ya que las referencias fantasmas están pensadas para acceder a la memoria cuando las referencias fuertes **ya no están disponibles**
    - El acceso a las referencias se realiza a través de la cola (`poll`)

```
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());
```

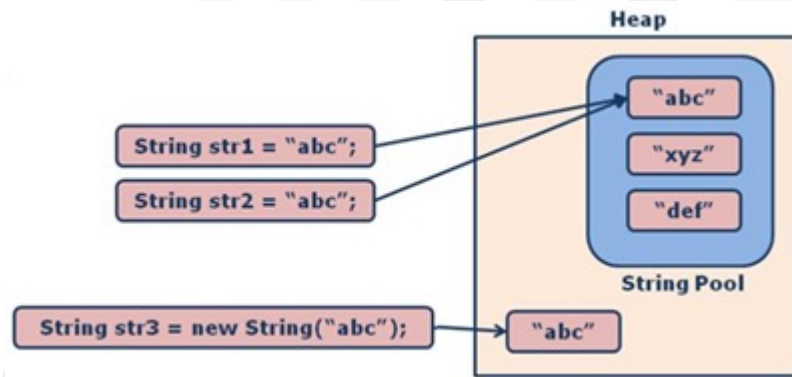
# Referencias: Gestión avanzada

- Usos principales de los diferentes tipos de referencias
  - Las referencias suaves y las referencias fantasma se utilizan para **hacer cachés en memoria**, de manera que se pueda acceder a las referencias fuertes que ya no están disponibles en memoria
  - Las referencias débiles se utilizan para acceder **dinámicamente** a la referencia fuerte de un objeto hasta que dicho objeto ya no esté disponible, evitando tener que crear referencias indiscriminadas del dicho objeto (*aliasing*)
    - **Ejemplo:** acceso a la conexión a una base de datos para hacer una consulta, de modo que la referencia débil se mantiene mientras el recolector de basura no la elimine, en cuyo caso será necesario realizar una nueva petición de conexión

# Referencias: Clase String

- Los objetos que son cadenas de texto se pueden crear de **dos** formas diferentes

- **Directamente:** asignando una cadena de texto al objeto, en cuyo caso se almacenan en una zona del montón llamada **String Pool**, de modo que **cada vez que se asigna la misma cadena de texto, se apunta a la dirección que la contiene** (str1 y str2)



- **Indirectamente:** el objeto se crea usando un constructor de String, en cuyo caso se almacenan en el motón, pero **fuera del String Pool**, aunque tenga el mismo valor que una cadena previa

# Referencias: Clase String

- **String** es una clase de Java, pero **no se comporta** como el resto de las clases en lo que respecta a la asignación entre objetos cuando se usan directamente cadenas de texto ("**<cadena>**")
  - Sigue funcionando igual cuando se realiza una asignación entre una cadena de texto  $CT_A$  y otra cadena de texto  $CT_B$  ( $CT_A = CT_B$ )

```
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

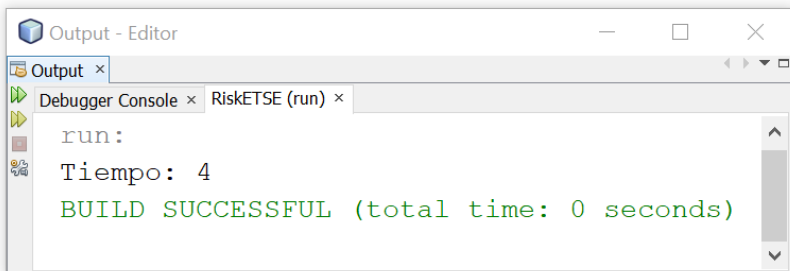
- En realidad una cadena de texto es **un objeto inmutable**, es decir, una vez se ha reservado memoria y se le asigna un valor dado, no se puede modificar



# Referencias: Clase String

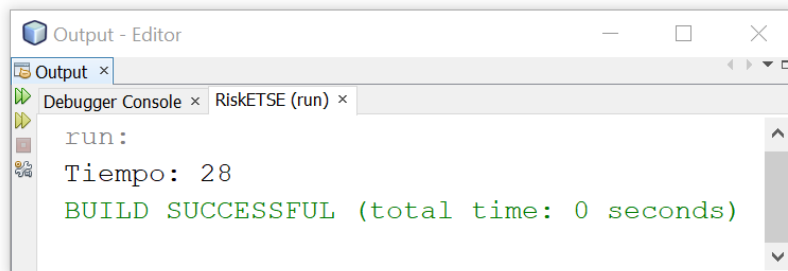
- El uso indiscriminado de cadenas de texto puede llevar a una **merma importante en el rendimiento** de un programa

```
public static void main(String[] args) {  
    long in= System.currentTimeMillis();  
    long x= 0;  
    long y= 0;  
    for(int i=0;i<100000;i++) {  
        x+= i;  
        y+= i;  
    }  
    long out= System.currentTimeMillis();  
    System.out.println("Tiempo: " + (out-in));  
}
```



The screenshot shows the 'Output - Editor' window with the 'Output' tab selected. It displays the output of the first program: 'run: Tiempo: 4' followed by 'BUILD SUCCESSFUL (total time: 0 seconds)'.

```
public static void main(String[] args) {  
    long in= System.currentTimeMillis();  
    long x= 0;  
    String y= "";  
    for(int i=0;i<100000;i++) {  
        x+= i;  
        y= "" + i;  
    }  
    long out= System.currentTimeMillis();  
    System.out.println("Tiempo: " + (out-in));  
}
```



The screenshot shows the 'Output - Editor' window with the 'Output' tab selected. It displays the output of the second program: 'run: Tiempo: 28' followed by 'BUILD SUCCESSFUL (total time: 0 seconds)'.

# Referencias

- Buenas prácticas de programación (X)

Si se van a realizar múltiples modificaciones sobre una cadena de texto, no se debería de utilizar la clase String



- Buenas prácticas de programación (XI)

Se debería usar la clase StringBuffer si se van a realizar múltiples operaciones sobre cadenas de texto, ya que no se reserva espacio de memoria cada vez que se genera una cadena de texto



# Referencias

En el siguiente trozo de código la clase `Coche` tiene un atributo denominado `matricula` de tipo `String` y un método llamado `getMatricula()` que obtiene el valor de dicho atributo. Teniendo esto en cuenta, ¿cuáles de las siguientes afirmaciones son ciertas?

```
1  ArrayList<Coche> coches= new ArrayList<>();
2  coches.add(new Coche("0000 ABC"));
3  coches.add(new Coche("0001 DEF"));
4  coches.add(new Coche("0002 HIJ"));
5  coches.add(new Coche("0004 KLM"));
6  Coche coche1= coches.get(3);
7  String mat3= coche3.getMatricula();
8  mat3= "0003 HIJ";
9  for(int i=0;i<coches.size();i++) {
10     if(i==3) coches.remove(i);
11     else {
12         String mensaje= "Coche no eliminado";
13         System.out.println(coches.get(i).getMatricula());
14     }
15 }
```

- ☐ Después de que se haya ejecutado la línea 8, el valor de la matrícula del último coche que se ha introducido es "0004 KLM".
- ☐ En la línea 7 tiene lugar aliasing.
- ☐ En la línea 6 tiene lugar aliasing.
- ☐ En todas las iteraciones del bucle `for` se realizan tres reservas de memoria para objetos de tipo `String`.
- ☐ Entre las líneas 6 y 8 (incluidas), se realizan dos reservas de memoria para objetos de tipo `String`.
- ☐ La ejecución del bucle `for` genera un error porque se accede a una posición de la lista en la que no hay ningún objeto, es decir, se accede a un `null`.

# Identidad de objetos: Método equals

- ¿Cuándo se puede considerar que dos objetos son iguales?
- Opción 1: dos objetos son iguales si ocupan la misma posición de memoria
  - Esta condición es **muy restrictiva**, ya que en realidad no se están comparando dos objetos, sino que se comparan dos referencias a un mismo objeto

```
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

# Identidad de objetos: Método equals

- ¿Cuándo se puede considerar que dos objetos son iguales?
- Opción 2: dos objetos son iguales si son del mismo tipo y si los valores de todos los atributos de los dos objetos son también iguales
  - Esta condición 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**

```
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 coincidiera

# Identidad de objetos: Método equals

- equals es un método que indica si un objeto, que es el argumento del método, **es igual** al objeto desde el que se invoca equals

```
@Override  
public boolean equals(Object object)
```

- equals 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 equals que tiene la clase **Object**, que es la clase que está en el nivel más alto de la jerarquía de Java

# Identidad de objetos: Método equals

- La implementación que se hereda de Object sigue la opción 1, es decir, compara las referencias de los dos objetos, ya que lo único que se conoce de cualquier objeto es su referencia

```
public boolean equals(Object obj)
{
    return this == obj;
}
```

- Es necesario reimplementar el método equals identificando los **atributos que son inmutables** una vez se reserva memoria para el objeto, de modo que dos objetos son iguales si los atributos inmutables tienen los mismos valores
  - Típicamente los objetos inmutables son **cadenas de texto** o **wrappers** asociados a tipos primitivos

# Identidad de objetos: Método equals

@Override

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

```
    if (this == obj) {  
        return true;  
    }
```

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

```
    if (obj == null) {  
        return false;  
    }
```

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

```
    if (getClass() != obj.getClass()) {  
        return false;  
    }
```

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

```
    final Jugador other = (Jugador) obj;  
    if (!this.nombre.equals(other.nombre)) {  
        return false;  
    }  
    if (!this.color.equals(other.color)) {  
        return false;  
    }
```

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)

```
    return true;
```

```
}
```



# Identidad de objetos: Método equals

- Buenas prácticas de programación (XII)

Todos los objetos con los que se realizan comparaciones deben de tener implementado el método equals



- Buenas prácticas de programación (XIII)

Los atributos que se usan en equals para definir la igualdad entre objetos deberían ser de tipo primitivo, wrappers a tipos primitivos o cadenas de texto

