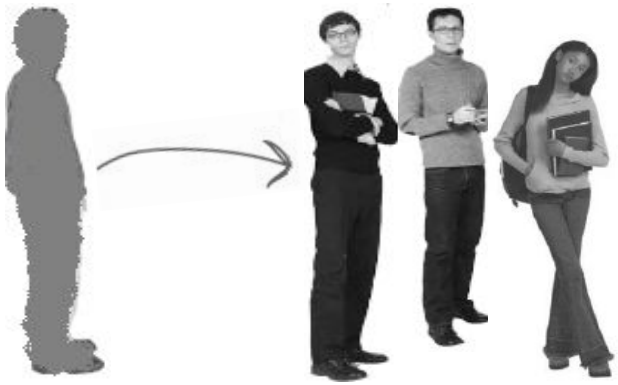
Clases e Instancias

- **Definición de una clase java**
  - Variables y métodos de instancia
  - Variables y métodos de clase
  - Tipos de variables: referenciales y primitivas
- **Instanciación una clase java**
  - El operador `new()`
  - Constructores

Clases e instancias

Una clase es un molde a partir del cual se crean instancias con las mismas características y comportamiento.

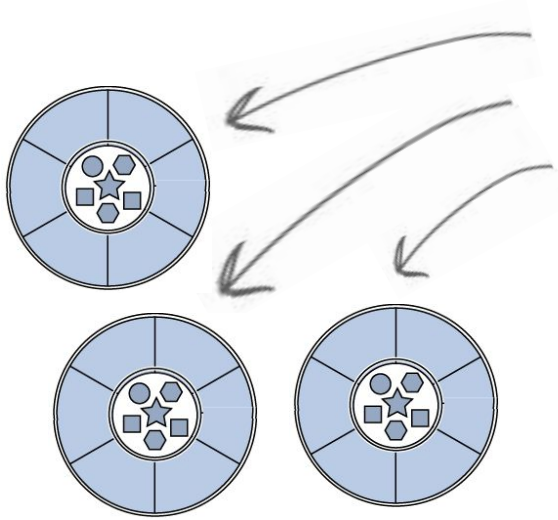


María y otras personas que hacen uso del ascensor fueron modelados como **instancias** de la clase Usuario (del ascensor).

- Una instancia u **objeto es una entidad de software** que combina un estado/datos y comportamiento/métodos.
- Cada instancia de una clase (objeto) tiene una copia de las variables de instancia y dispone de los métodos declarados en la clase.

¿Cómo definir/declarar una clase en java?

Una clase es un bloque de código o molde, que describe cómo serán los objetos que pertenecen a ella. Contiene variables que representan el estado de los objetos y métodos que representan los mensajes que entienden tales objetos. Un archivo origen java debe guardarse con el mismo nombre que la clase (y con extensión `.java`). Se deben respetar las mayúsculas.



objetos Jugador

Jugador.java

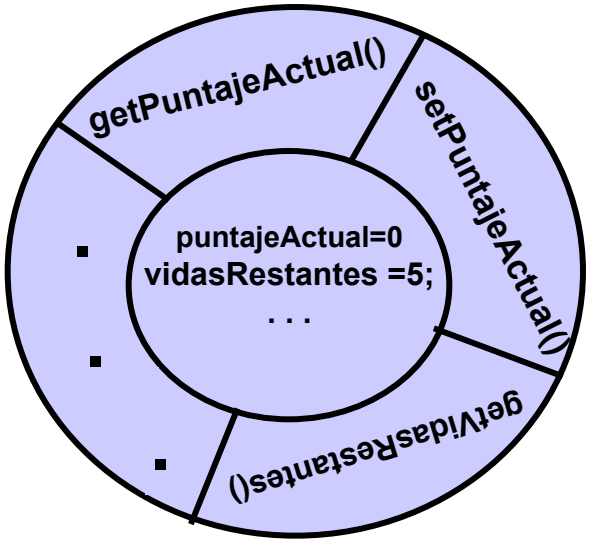
```
package juego;
public class Jugador {
 private int puntajeActual;
 private int vidasRestantes;
 public int getPuntajeActual() {
  return puntajeActual;
 }
 public void setPuntajeActual(int puntajeActual) {
  this.puntajeActual = puntajeActual;
 }
 public int getVidasRestantes() {
  return vidasRestantes;
 }
 public void setVidasRestantes(int vidasRestantes) {
  this.vidasRestantes = vidasRestantes;
 }
}
```

¿Cómo incorporar estado y comportamiento a una clase?

Comúnmente una clase contiene:

- **variables de instancia:** constituyen el estado de un objeto. Normalmente, las variables de instancia se declaran **`private`**, lo que significa que sólo la clase puede acceder a ellas directamente.
- **métodos de instancia:** definen las operaciones que pueden realizar los objetos de un tipo de clase. Un método es un bloque de código, similar a lo que es una función o procedimiento en los lenguajes procedurales, como PASCAL.

objeto Jugador



Jugador.java

```
package juego;
public class Jugador {
  private int puntajeActual;
  private int vidasRestantes;
  public int getPuntajeActual() {
    return puntajeActual;
  }
  public void setPuntajeActual(int puntajeActual) {
    this.puntajeActual = puntajeActual;
  }
  public int getVidasRestantes(){
    return vidasRestantes;
  }
}
```

Estado
(variables de instancia)

Tipo de retorno. Si el método no devuelve ningún valor, se utiliza la palabra `void`.

Comportamiento
(métodos de instancia)

¿Cómo incorporar estado y comportamiento a una clase?

En java cada método y cada variable, existen adentro de una clase: java no soporta funciones o variables globales.

La declaración de una variable de instancia debe incluir:

- Un identificador (nombre de la variable).
- Un tipo (tipo primitivo o de un tipo de una clase).
- Un modificador de acceso (opcional): **`public`** o **`private`**.

La declaración de un método de instancia debe especificar:

- Un nombre
- Una lista de argumentos (opcional)
- Un tipo de retorno
- Un modificador de acceso (opcional): **`public`** o **`private`**.

```
package juego;
public class Jugador {
 private int puntajeActual;
 private int vidasRestantes;
 public int getPuntajeActual() {
  return puntajeActual;
 }
 public void setPuntajeActual(int puntajeActual) {
  this.puntajeActual = puntajeActual;
 }
 public int getVidasRestantes() {
  return vidasRestantes;
 }
 public void setVidasRestantes(int vidasRestantes){
  this.vidasRestantes = vidasRestantes;
 }
}
```

No tiene importancia el orden en que se ubican las variables y los métodos.

Firma o encabezado del método

Tipos de datos en Java

En java hay 2 categorías de tipos de datos: tipo primitivo y tipo de una clase (referencia).

- **Tipos primitivos**: las variables de tipo primitivo mantienen valores simples y NO son objetos. Existen 8 tipos de datos primitivos:

Entero: **`byte, short, int, long`**
Punto flotante: **`float y double`**
Un carácter de texto: **`char`**
Lógico: **`boolean`**

Declaración e inicialización de variables primitivas

```
float pi = 3.14;
double saldo = 0;
char letra = 'A';
int hora = 12;
boolean es_am = (hora>12);
```

- **Tipos de una clase:** las variables que referencian a un objeto son llamadas *variables referencias* y contienen la ubicación (dirección de memoria) de objetos en memoria.

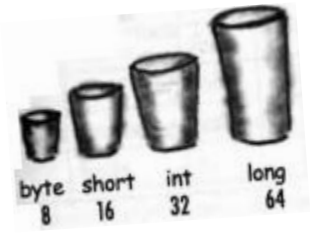
Declaración e inicialización de variables referencias

```
Jugador jugador;                    Fecha diaCumple = new Fecha();
jugador = new Jugador();
```

Tipos de datos en Java
Inicialización

Si la definición de una clase no inicializa variables de instancia, las mismas toman valores por defecto.

- Las variables de instancia de **tipo primitivo** se inicializan con los siguientes valores por defecto:



Tipo primitivo	Valor por defecto
boolean	false
char	'\uoooo' (nulo)
byte/short/int/long	0
float/double	0.0

- Las variables de instancia que son referencias a objetos, se inicializan con el valor por defecto: **`null`**.

Nota: las variables locales, es decir, las variables declaradas dentro de un método, deben inicializarse explícitamente antes de usarse.

Tipos de datos en Java
Clases Wrapper

- Java no considera a los tipos de datos primitivos como objetos. Los datos numéricos, booleanos y de caracteres se tratan en su forma primitiva por razones de eficiencia.
- Java proporciona clases *wrappers* (o también conocidas como *primitive-boxed*) para manipular a los datos primitivos como objetos. Los datos primitivos están envueltos (*"wrapped"*) en un objeto que se crea entorno a ellos.
- Cada tipo de datos primitivo de Java, posee una clase *wrapper* correspondiente en el paquete `java.lang`. Cada objeto de la clase *wrapper* encapsula a un único valor primitivo.

Tipo primitivo	Clase Wrapper
`char`	`Character`
`boolean`	`Boolean`
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`

Tipos de datos en Java
Clases Wrapper

Antes de la versión J2SE 5.0 de Java, se debía convertir de tipos primitivos a objetos y viceversa explícitamente usando métodos o el operador new.

```
Integer nro = new Integer(3);
int num = nro.intValue();
Character letra = new Character('a');
char l= letra.charValue();
```

A partir de esta versión, se dispone de conversiones automáticas de tipos primitivos a objetos y viceversa. Se pueden hacer asignaciones automáticas entre tipos. Esta facilidad se conoce como autoboxing.

```
Integer nro = 3;
Character letra = 'a';
int num = nro;
```

Si bien ahora las diferencias entre los tipos primitivos y las clases wrappers asociadas son más imperceptibles, no desaparece la distinción entre ellos. Por ejemplo pensemos en un código que calcule la sumatoria de los números enteros desde 0 hasta el entero más grande. Acumulemos primero el valor en un objeto de tipo `Long` y luego en un primitivo `long`. ¿Cuál es el impacto?

```
Long suma = 0L;
long antes = System.currentTimeMillis();
for (int i = 0; i < Integer.MAX_VALUE; i++){
   suma += i;
}
```

tarda: 37.66 seg.

```
long suma = 0L;
long antes = System.currentTimeMillis();
for (int i = 0; i < Integer.MAX_VALUE; i++){
    suma += i;
}
```

tarda: **5.526 seg.**

¿Cómo se instancia una clase?

Para instanciar una clase, es decir, para crear un objeto de una clase, se usa el operador **new**. La creación e inicialización de un objeto involucra los siguientes pasos:

a) Se aloca espacio para la variable

b) Se aloca espacio para el objeto en la HEAP y se inicializan los atributos con valores por defecto.

c) Se inicializan explícitamente los atributos del objeto.

d) Se ejecuta el constructor (*parecido* a un método que tienen el mismo nombre de la clase)

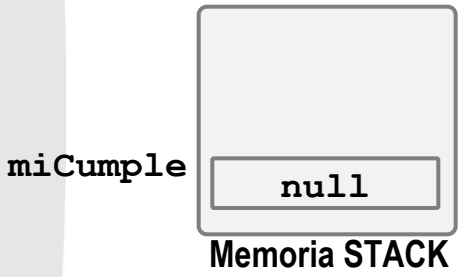
f) Se asigna la re

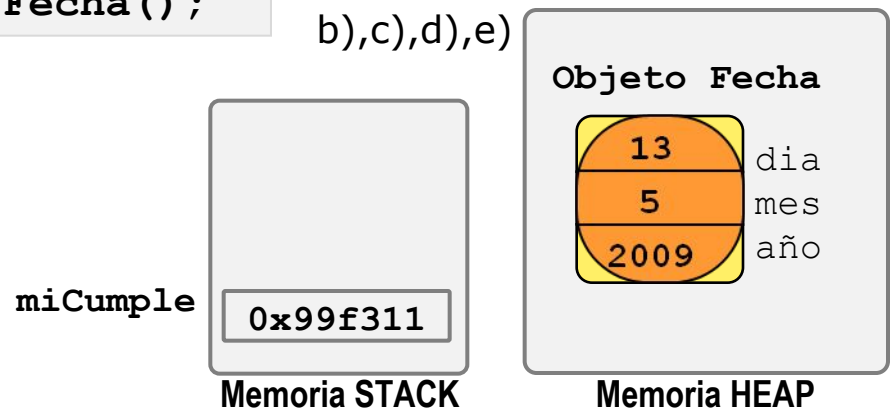
```
public class Fecha {
 private int dia = 13;
 private int mes = 5;
 private int año = 2009;
 // métodos de instancia
}
```

a)

```
Fecha miCumple;
miCumple = new Fecha();
```

b),c),d),e)





¿Cómo manipulo los datos de un objeto?

¿Cómo manipular el objeto?

Una vez que se ha creado un objeto, seguramente es para usarlo: cambiar su estado, obtener información o ejecutar alguna acción. Para poder hacerlo se necesita:

- conocer la **variable referencia**

```
package modelo;
public class Jugador {
 private String nombre;
 private int edad;
 private Rol rol;
 private Casillero posicionActual;
 public String getNombre() {
  return nombre;
 }
 public void setNombre(String nombre) {
  this.nombre = nombre;
 }
 public String getRol() {
  return rol;
 }
 public void setRol(String rol) {
  this.rol = rol;
 }
. . .  //otros getter/setter
}
```

Instanciación de un objeto Jugador e invocación de sus métodos

```
package tallerII;
import modelo.Jugador;

public class JugadorTest{
 public static void main(String[] args){
   Jugador jugador = new Jugador();
   jugador.setRol("Caballero");
   String s = jugador.getRol();
 }
}
```

Nota: Se recomienda declarar todos los atributos privados y utilizar métodos públicos para acceder al estado.

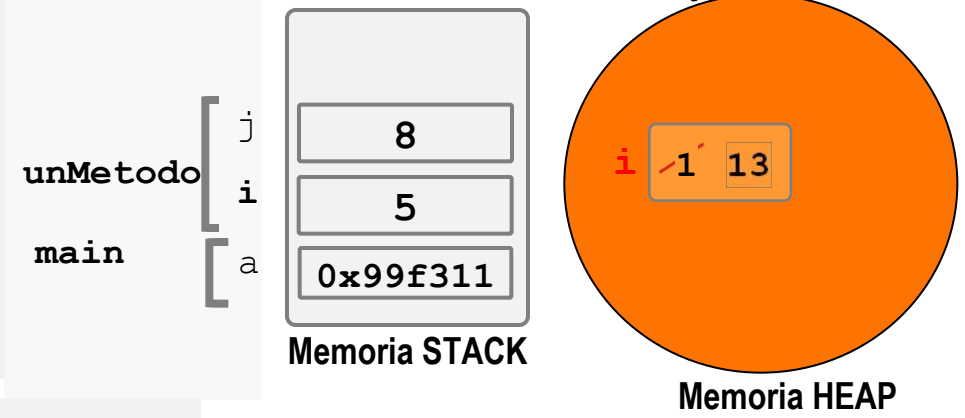
Variables de instancia y variables locales

Además de poder definir a una variable de tipo primitivo o referencia, es posible declararlas en dos lugares diferentes (siempre adentro de la clase):

- **afuera de cualquier método**. Son las variables de instancia que son creadas cuando el objeto es construido y existen mientras exista el objeto.
- **adentro de un método**. Estas variables son llamadas variables locales y deben inicializarse antes de ser usadas. Los parámetros de los métodos también son variables locales y las inicializan el código que llama al método. Estas variables son creadas cuando uidas cuando el método finaliza su ejecución.

```
public class AlcanceVariables {
 private int i=1;
 public void unMetodo(int i){
    int j=8;
    this.i=i+j;
 }
}
```

Si no usamos el this la asignación no tiene efecto




```
public class Test {
public static void main(String[] args) {
  AlcanceVariables a = new AlcanceVariables();
  a.unMetodo(5);
 }
}
```

this es una referencia al objeto actual. Está disponible automáticamente en todos los métodos

Variables y métodos de clase
La palabra clave `static`

- La palabra clave **`static`** permite definir variables y métodos de clase.

  Las variables de clase son **compartidas** por todas las instancias de la clase y puede accederse a ellas a través del nombre de la clase. No es necesario crear instancias. Si una clase tienen declarada una variable private static in **`ultJugador`**; donde se registra la cantidad de jugadores a los que se les asignó un Id, se puede consultar directamente a

  ```
  Jugador.getUltJugador();
  ```

- Un método de clase solo tiene acceso a sus variables locales, parámetros y variables de clase y no tiene acceso a las variables de instancia. Por qué?

- Algunos ejemplos de uso de **`static`**

  - En la API de JAVA la clase Math utiliza la palabra clase **`static`** para mantener por ejemplo el valor de PI y para declarar la mayoría de sus método. Por qué?

```
public final class Math {
 public static final double PI = 3.14159265358979323846;

 public static double tan(double arg0){ . . }
 public static double cos(double arg0){ . . }
 . . .
}
```

s declarado **`static`**. Por qué?

```
public class TestJugador {
 public static void main(String[] a){
  Jugador j1 = new Jugador();
  . . .
 }
}
```

Variables y métodos de clase

La palabra clave `static`

La palabra clave `static` declara atributos (variables) y métodos asociados con la clase en lugar de asociados a cada una de las instancias de la clase.

Las variables de clase son **compartidas** por todas las instancias de la clase. En el caso de los métodos de clase se utilizan cuando se necesita algún comportamiento que no depende de una instancia particular. En ambos casos se debe anteponer la palabra clave `static` al tipo de dato de la variable o de retorno del método.

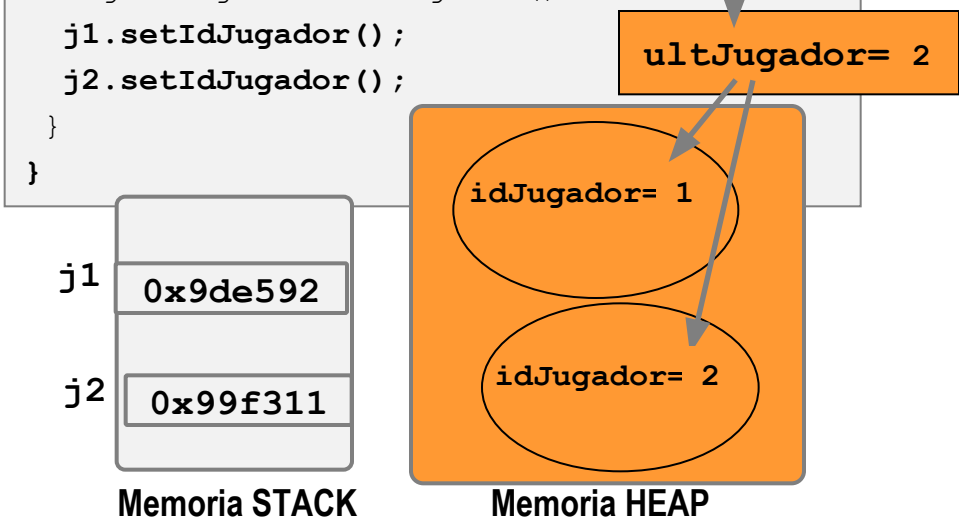
La variable **ultJugador** es compartida por todas las instancias de la clase Jugador

```
public class Jugador {
 private static int ultJugador = 0;
 private int idJugador;
 public void setIdJugador() {
   ultJugador++;
   this.idJugador = ultJugador;
 }
 public static int getUltJugador(){
   return ultJugador;
 }
  . . .  //otras variables y métodos
}
```

`Jugador.getUltJugador()` podria invocarse desde cualquier lugar aún no habiéndose creado instancias de Jugador

ultJugador es accesible desde cualquier instancia de Jugador

```
public class TestJugador {
 public static void main(String[] a){
  Jugador j1 = new Jugador();
  Jugador j2 = new Jugador();
  j1.setIdJugador();
  j2.setIdJugador();
 }
}
```



¿Qué son los Constructores?

- Los **constructores** son piezas de código (sintácticamente similares a los métodos) declaradas en el cuerpo de una clase, que permiten definir el estado inicial de un objeto en el momento de su creación.
- Los constructores son invocados automáticamente cuando se crea un objeto con el operador new.
- Los constructores se diferencian de los métodos porque:
  - Tienen el mismo nombre que la clase. La regla que indica que el nombre de los métodos debe comenzar con minúscula, no se aplica a los constructores.
  - No retornan un valor.
  - Son invocados automáticamente.

```
public class Vehiculo {
  private String marca;
  private double precio;

  public Vehiculo() {
  }
}
```

NO retorna valor

La inicialización está garantizada: cuando un objeto es creado, se aloca almacenamiento en la memoria HEAP y se invoca al constructor.

Nota: la expresión **new** retorna una referencia a un objeto creado recientemente, pero el constructor no retorna un valor.

Constructor sin argumentos

Un constructor sin argumento o constructor *nulo* es usado para crear un objeto básico.

- Si una clase NO declara constructores, el compilador inserta automáticamente un constructor nulo, con cuerpo vacío en el archivo .class.

```
public class Vehiculo {
  private String marca;
  private double precio;

  //métodos
}
```

compilador

```
public  Vehiculo(){

}
```

Cuando se crea un objeto de la clase **Vehiculo** con **new Vehiculo()** se invocará el constructor nulo, aún cuando no se haya declarado explícitamente.

- Si la clase define al menos un constructor con o sin argumentos, el compilador **NO insertará nada**.

Constructores con argumentos

Los constructores son usados para inicializar el estado del objeto que se está creando. Para especificar los valores para la inicialización se utilizan los parámetros del constructor.

```
public class Jugador {
 private String nombre;
 private int edad;
 private Rol rol;
 private Casillero posicionActual;

 public Jugador(String nombre, String rol) {
   this.nombre = nombre;
   this.rol = rol;
 }
 . . .
}
```

Si no usamos el **this** la asignación no tiene efecto

Codifcaciones equivalentes

```
public Jugador(String n_jug, String  r_jug){
  nombre = n_jug;
  rol = r_jug;
}
```

Si este constructor es el único de la clase, el compilador no permitirá crear un objeto **Vehiculo** de otra manera que no sea usando este constructor. Ejemplos:

```
Jugador j1 = new Jugador("Loren", "Caballero");
Jugador j2 = new Jugador("Spick", "Ladrón");
Jugador j3 = new Jugador();
```

El operador **new()** se puede utilizar en cualquier lugar del código.

Sobrecarga de Constructores

¿Es posible construir un objeto Jugador de distintas maneras?

Si, para ello se deben escribir en la clase más de un constructor. Esto es conocido como sobrecarga de constructores.

```
public class Jugador {
 private String nombre;
 private int edad;
 private String rol;
 private Casillero posicionActual;
 public Jugador() {
  this.nombre = "anonimo";
 }
 public Jugador(String nombre) {
  this.nombre = nombre;
 }
 public Jugador(String nombre,String rol) {
  this.nombre = nombre;
  this.rol = rol;
 }
}
```

La sobrecarga de constructores permite disponer de diferentes maneras para inicialización de los objetos de una clase.

```
public class TestJugadores{
 public static void main(String[] args){
  Jugador j1= new Jugador("Loren","Caballero");
  Jugador j2= new Jugador("Spick");
  Jugador j3= new Jugador();
 }
}
```

De la misma manera que lo hacemos con los constructores, es posible definir en una clase, varios métodos con el mismo nombre. Por ejemplo la clase **Math** del paquete **java.lang** define diferentes versiones del método **abs**

```
public final class Math {
 public static int abs(int a) { . . .}
 public static long abs(long a) { . . . }
 public static float abs(float a) { . ..}
 . . .
}
```

this() y this

Java pone disponible para el programador dos usos diferentes de la palabra clave **this**: uno para hacer referencia al objeto actual (**this**) y otro para ser usado desde un constructor para invocar a otro constructor de la misma clase (**this()**).

```
public class Jugador {
 private String nombre;
 . . .
 public Jugador() {
   this.nombre = "anonimo";
 }
 public Jugador(String nombre){
   this.nombre = nombre;
 }
 public Jugador(String nombre, String rol){
   this(nombre);
   this.rol = rol;
 }
 public void setRol(String rol) {
   this.rol = rol;
 }
}
```

this() se debe ubicar en la primera línea e invoca al constructor de la misma clase con un String como parámetro.

this()

Cuando en una clase, hay más de un constructor, puede surgir la necesidad de invocarse entre ellos para evitar duplicar código.

this

La palabra clave **this** mantiene una referencia al objeto "actual", está disponible automáticamente adentro del cuerpo de los métodos de instancia y de los constructores. A través del **this** es posible manipular variables de instancia e invocar a métodos de instancia, de la misma manera que se usa cualquier variable que referencia a un objeto.

```
public class TestJugador {
public static void main(String[] args) {
   Jugador j1 = new Jugador("Fran");
   Jugador j2 = new Jugador("Slash");
   j1.setRol("Caballero");
   j2.setRol("Ladron");
 }
}
```

J1 viaja como parámetro oculto.

J2 viaja como parámetro oculto

Sean j1 y j2 dos variables que referencian a dos objetos diferentes de tipo Jugador. Si invocamos el método **setRol(String rol)** sobre ambos objetos, ¿cómo sabe el método **setRol** que variable **rol** actualizar?

Sobrecarga de Constructores

¿Es posible manejar la variable `ultJugador` desde los constructores?

Si, se puede. Se debe ser cuidadoso para contar solamente una vez al jugador creado.

```
package modelo;
public class Jugador {
 private String nombre;
 private int edad;
 private String rol;
 private Casillero posicionActual;
 private static int ultJugador = 0;

 public Jugador() {
  this.nombre = "anonimo";
  idJugador=++ultJugador;
 }
 public Jugador(String nombre) {
  this.nombre = nombre;
  idJugador=++ultJugador;
 }
 public Jugador(String nombre, String rol) {
  this(nombre);
  this.rol = rol;
 }
 . . .
}
```

```
public class TestJugadores{
  public static void main(String[] args){
    Jugador j1 = new Jugador("Loren", "Caballero");
    Jugador j2 = new Jugador("Spick");
    Jugador j3 = new Jugador();
          System.out.println("El último Id es: "+
Jugador.getUltJugador());
  }
}
```

```
El último Id es: 3
```

Acá no hace falta incrementar porque se invoca a otro constructor que lo hace

Objetos de tipo String

Un `String` es una secuencia de caracteres (valores `char`). En java un `String` es un objeto. Los `String` son ampliamente usados en cualquier tipo de aplicación.

Creación

La manera mas directa de crear un String es escribirlo así:

```
String s1 = “Hola Mundo”;
```

En este caso es un *literal string*, una secuencia de caracteres entre comillas

Un objeto String tambien puede crease como cualquier objeto:

```
String s2 = new String(“Hola Mundo”);
```

¿Cuál es la salida de estas impresiones?

```
String str1 = "Hola Mundo!";
String str2 = "Hola Mundo!";
String str3 = new String("Hola Mundo!");
System.out.println(str1==str2);
System.out.println(str1==str3);
System.out.println(str1.equals(str3));
```

Los literales con el mismo contenido comparten un pool de String

`true`

`false`

`true`

Concatenación

Los `Strings` pueden concatenarse usando `+` o el método `concat()`. Debido a que los `Strings` son inmutables, siempre retorna un nuevo `String`.

```
String hola = "Hola";
String saludo1 = hola.concat(" Mundo");
String saludo2 = hola + " Mundo";
System.out.println(hola); System.out.println(saludo1); System.out.println(saludo2);
```

```
Hola
Hola Mundo
Hola Mundo
```

Objetos de tipo String

Formatos para String

El método `printf()` permite imprimir un `String` formateado en la salida estándar:

```
System.out.printf("El valor de la variable " +
                  "float es %f, mientras que " +
                  "el valor de la variable " +
                  "int es %d, " +
                  "y el String es %s",
                  floatVar, intVar, stringVar);
```

```
El valor de la variable float es
3,141600, mientras que el valor de
la variable int es 3, y el String
es Hola
```

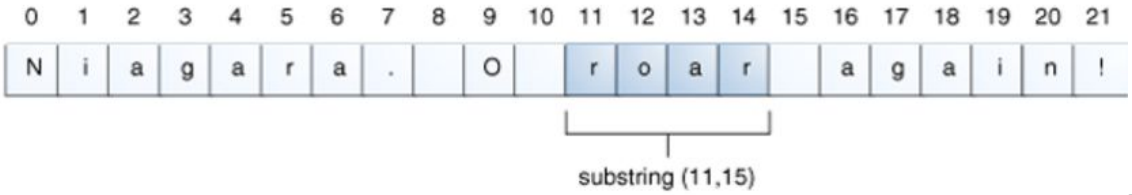
Se esperan tres variables: una de tipo float (%float), un entero (%d) y un String. (%s)

El método estático `format()` permite crear un `String` formateado que se puede reutilizar, como opuesto a la sentencia `print()` que lo arma para imprimir.

```
String fs;
fs = String.format("El valor de la variable" +
        "float es %f, mientras que " +
        "el valor de la variable int es %d, " +
        "y el String es %s", floatVar, intVar, stringVar);
System.out.println(fs);
```

Substring

```
String anotherPalindrome = "Niagara. O roar again!";
String roar = anotherPalindrome.substring(11, 15);
```



String, StringBuffer y StringBuilder

Además de la clase **String**, existen dos clases muy similares **StringBuffer** y **StringBuilder** para manipular cadena de caracteres. Estas clases mantienen una cadena de caracteres mutable. La única diferencia entre estas clases es que los métodos de **StringBuffer** son sincronizados, por lo cual la podemos usar de manera segura en un ambiente de multihilos. Los métodos de **StringBuilder** no son sincronizados, por lo que tiene mejor rendimiento que **StringBuffer**.

```
StringBuffer str = new StringBuffer();
str.append("Hola,");
str.append("mundo");
```

```
StringBurilder str = new StringBuilder();
str.append("Hola,");
str.append("mundo");
```

¿Es más rápido un **StringBuilder**? A modo de ejemplo, vamos a concatener un millón de strings "java" y compararemos los tiempos.

```
public static void main(String[] args) {
    StringBuffer sbuffer = new StringBuffer();
    long inicio = System.currentTimeMillis();
    for (int i = 0; i < 1000000; i++) {
        sbuffer.append("java");
    }
    long fin = System.currentTimeMillis();
    System.out.println("Tiempo del StringBuffer: " + (fin - inicio));

    StringBuilder sbuilder = new StringBuilder();
    inicio = System.currentTimeMillis();
    for (int i = 0; i < 1000000; i++) {
        sbuilder.append("java");
    }
    fin = System.currentTimeMillis();
    System.out.println("Tiempo del StringBuilder: " + (fin - inicio));
}
```

```
Tiempo del StringBuffer: 60
Tiempo del StringBuilder: 27
```

Un **StringBuilder** puede resultar un 50% más rápido para concatenar Strings.

Nota: con este mismo código usando String (+), dio:

Tiempo del String: 2857212