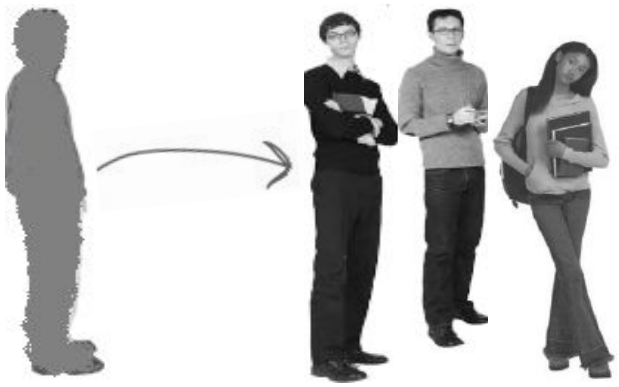
Clases, Instancias y paquetes

- **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
- **Paquetes**
  - Definición de paquetes
  - Archivos JAR

Clases e instancias

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

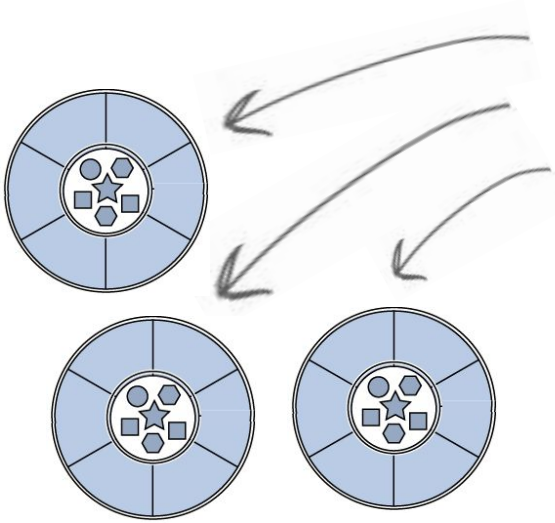


María y otras personas que hacen uso del ascensor fueron modelados como **instancias** de la clase Usuario (del ascensor).

- 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 de tipo Jugador

Jugador.java

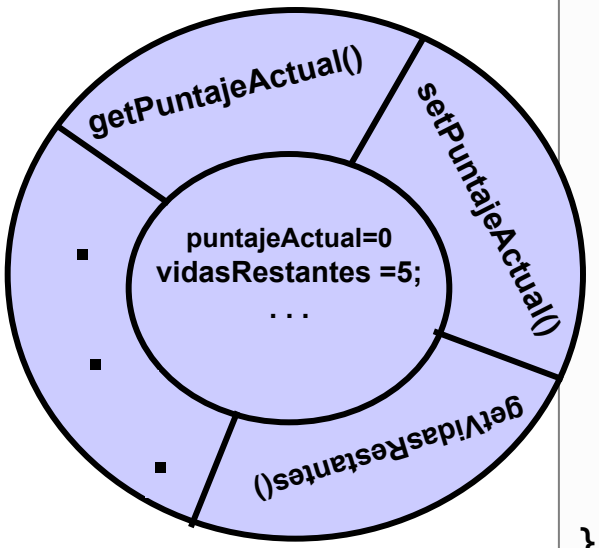
```
package juego;
public class Jugador {
 private int puntajeActual;
 private int vidasRestantes;

 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, identifican los datos almacenados en cada objeto)

Comportamiento
(métodos de instancia)

¿Cómo incorporar estado y comportamiento a una clase?

En java los métodos y las variables, 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.

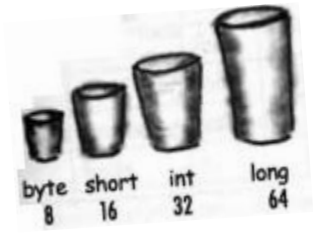
Declaración e inicialización de variables referencias

```
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. Java permite tratar a los datos numéricos, booleanos y de caracteres 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 en torno 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

Autoboxing

Es la conversión automática que realiza el compilador de Java entre los tipos primitivos y sus clases wrappers correspondientes. Por ejemplo, convertir un int en un Integer, un double en un Double, etc.

```
Character c = 'a';
Integer i = 7;
```

Unboxing

Es la conversión es al revés, es decir conversión de wrapper a un primitivo. Por ejemplo de un Character a char o Double a double.

```
char c1 = c;
int i1 = i;
```

Estos ejemplos usan tipos genéricos y colecciones que veremos más adelante:

```
List<Integer> li = new ArrayList<>();

for (int i = 1; i < 50; i += 2)
    // 1. Autoboxed al invocar el método
    li.add(i);
```

```
List<Double> lista = new ArrayList<>();
// 1. Autoboxed al invocar el método
lista.add(3.1416);

// 2. Unbox al asignar
double pi = lista.get(0);
System.out.println("pi = " + pi);
```

Cuándo es recomendable usar primitivos?

```
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;
}
```

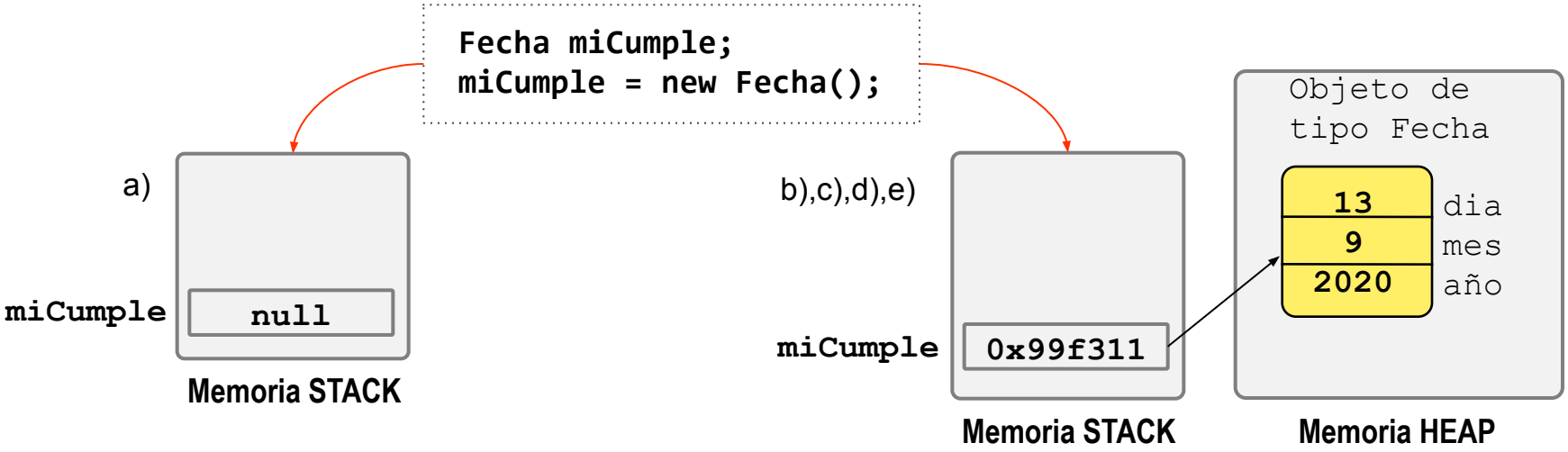
tarda: **5.526 seg.**

¿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)
e. Se asigna la referencia del nuevo objeto a la variable.

```
public class Fecha {
  private int dia = 13;
  private int mes = 9;
  private int año = 2020;
  // métodos de instancia
}
```



¿Cómo se manipula el objeto?

¿Cómo se manipula un 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**
- utilizar el operador **"."**

```
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.

¿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;

                                    NO retorna valor
 public Vehiculo() {
 }
}
```

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 **Jugador** 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(xxx)` se debe ubicar en la primera línea del constructor donde se usa.

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?

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.

  Por ejemplo, si una clase tiene declarada una variable:

  `private static int` `ultJugador`; se podría consultar directamente a la clase:

  `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?
  - El método **`main()`** que venimos utilizando también es declarado **`static`**. 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){ . . }
 . . .
}
```

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

Sobrecarga de Constructores

Supongamos que declaramos una variable de clase `ultJugador` para mantener la cantidad de instancias creadas a partir de una clase. Se puede manipular desde los constructores?

```
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 static int getUltJugdor(){
  return ultJugador;
 }
 . . .
}
```

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

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

```
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: "+
          ugador.getUltJugador());
  }
}
```

```
El último Id es:3
```

La clase 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 más 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 también puede crearse 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
```

La clase 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, como alternativa 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);
```

Un índice de argumento del **`format()`** se especifica como un número que termina con un **"$"** después del **"%"**, en ese caso selecciona el argumento especificado en la lista de argumentos.

```
String fs;
fs = String.format("%3$s", "Hello Laura", "Hola Laura", "Ciao Laura";
```

```
Ciao Laura
```

La clase String

La clase String también cuenta con el método **join** que permite convertir un objeto Iterable (como List) de Strings en un único String con los valores separados por un *character* pasado por parámetro. No agrega el delimitador al final de la cadena.

```
List<String> cities = Arrays.asList("Milan","London","New York","San Francisco");
String citiesCommaSeparated = String.join(",", cities);
System.out.println(citiesCommaSeparated);
```

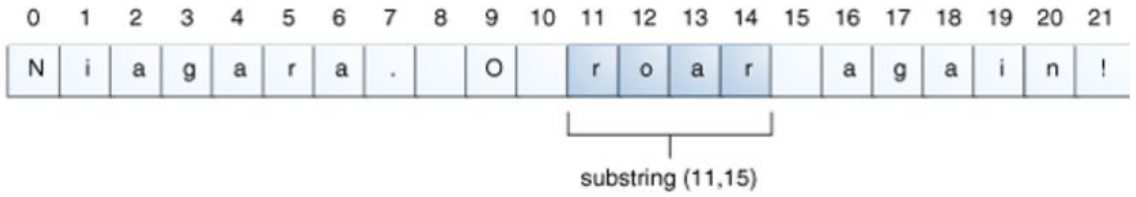
```
Milan,London,New York,San,Francisco
```

```
String baseDirectory = "home";
String subFolder = "qatar2022";
String countrysubFolder = "Argentina";
String fileName = "medallas.csv";
List<String>filePathParts=Arrays.asList(baseDirectory,subFolder,countrysubFolder,fileName);
File file = new File(String.join(File.separator, filePathParts));
System.out.println(file.getPath());
```

```
home/qatar2022/Argentina/medallas.csv
```

Para extraer una parte del string se lo puede hacer directamente sobre una instancia de String con el método **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));
}
```

Ambos paquetes contienen la clase Vector

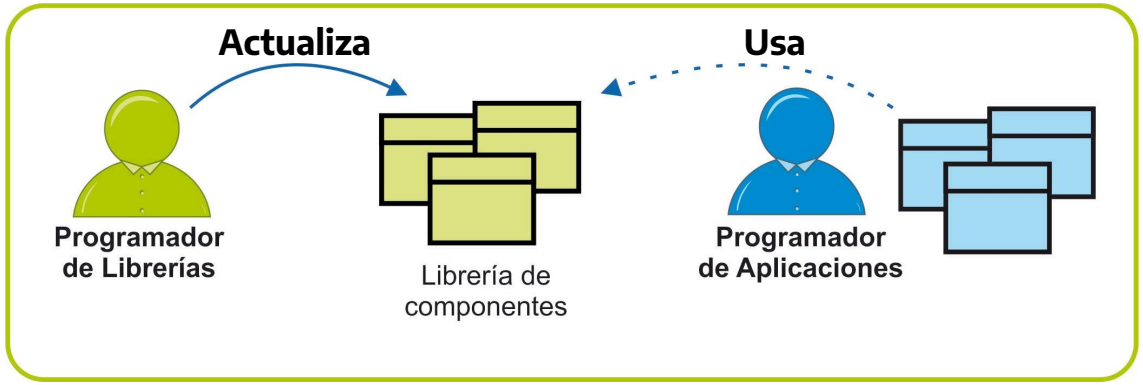
```
Tiempo del StringBuffer: 60
Tiempo del StringBuilder: 27
```

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

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

Tiempo del String: 2857212

¿Qué podría pasar si se modifica una librería de clases que está siendo usada por otros programadores?



El código podría romperse !!

El **programador de la librería** debe sentirse libre para **mejorar el código** y el **programador de aplicaciones** debería poder **aplicar las mejores** sin necesidad de reescribir su código.

¿Cómo se asegura esto?

(1) **Garantizando compatibilidad con versiones previas**: no quitar métodos existentes en la versión previa.

(2) **Usando especificadores de acceso** para indicarle al programador de aplicaciones qué está disponible y qué no.

Antes de entrar en especificadores de acceso, falta responder una pregunta útil en este contexto: **¿cómo se crea una librería de clases en java?**

Librería de componentes
Paquetes JAVA

- En Java **una librería de componentes** (clases e interfaces) es un agrupamiento de archivos `.class`, también llamado **paquete.**
- Para **agrupar componentes** en un paquete, debemos anteponer la palabra clave **`package`** junto con el nombre del paquete al comienzo del archivo fuente de cada una de las clases o interfaces.

```
package graficos;
public class Rectangulo {
  //código JAVA
}
```

```
package graficos;
public interface Centrable {
  //código JAVA
}
```

La clase **Rectangulo** y la interface **Centrable** pertenecen al paquete **graficos**

- Las clases e interfaces que se crean **sin usar la sentencia `package`** se ubican en un paquete sin nombre, llamado ***default package***.

```
public class HolaMundo {
      //código JAVA
}
```

La clase **HolaMundo** pertenece al **paquete por defecto**

Usar paquetes propios o el *default package* ¿Qué opinan? ¿por qué?

Librería de componentes

Paquetes JAVA

El **nombre completo de la clase o nombre canónico** contiene el nombre del paquete.

```
graficos.Rectangulo
java.util.Arrays
```

Nombre completo de la clase Rectangle

Nombre completo de la clase Arrays

Para usar la clase **Rectangulo** se debe usar la palabra clave **`import`** o **especificar el nombre completo** de la clase:

```
package ar.edu.unlp.taller2;
import graficos.Rectangulo;
//import graficos.*;
class Figuras {
 Rectangulo r = new Rectangulo();
}
```

```
package ar.edu.unlp.taller2;

class Figuras {
 graficos.Rectangle r;
 r = new graficos.Rectangle();
}
```

Importación por demanda: se tienen disponibles todos los nombres de clases e interfaces del paquete

La sentencia **`import`** permite usar el **nombre corto de la clase** en todo el código fuente. Si no se usa el **`import`** se debe especificar el **nombre completo** de la clase.

Librería de componentes

Paquetes JAVA

¿Qué sucede si se crean 2 clases con el mismo nombre?

Supongamos que 2 programadores escriben una clase de nombre **Vector** en el paquete *default,* **se plantea un conflicto de nombres**.

Es necesario **crear nombres únicos: usamos paquetes.**

```
package util;
public class Vector {
   //código JAVA
}
```

```
package taller2.estructuras;
public class Vector {
    //código JAVA
}
```

¿Qué sucede si se importan dos librerías que incluyen el mismo nombre de clase?

```
import taller2.estructuras.*;
import util.*;

Vector vec1 = new Vector();
```

Ambos paquetes contienen la clase Vector

Colisión! ¿A qué clase hace referencia?: el compilador no puede determinarlo

Librería de componentes

Paquetes JAVA

```
package util;
public class Vector {
   //código JAVA
}
```

```
package taller2.estructuras;
public class Vector {
    //código JAVA
}
```

¿Qué sucede si se importan dos librerías que incluyen el mismo nombre de clase?

```
import taller2.estructuras.*;
import util.*;

Vector vec1 = new Vector();
```

Ambos paquetes contienen la clase Vector

Colisión! ¿A qué clase hace referencia?: el compilador no puede determinarlo

Una posible solución:

```
import util.*;
Vector vec1 = new Vector();

taller2.estructuras.Vector vec2 = new taller2.estructuras.Vector();
```

Usamos los dos nombres canónicos o combinamos

Librería de componentes
Paquetes JAVA

- Las clases e interfaces que son parte de la distribución estándar de JAVA están agrupadas en paquetes de acuerdo a su funcionalidad. Algunos paquetes son:

`java.lang`	clases básicas para crear aplicaciones.
`java.util`	librería de utilitarios, colecciones.
`java.io`	manejo de entrada/salida.
`java.awt /javax.swing`	manejo de GUI (Graphic User Interface).

- Los únicos paquetes que se importan automáticamente es decir no requieren usar la sentencia `import` son el paquete `java.lang` y el **paquete actual** (paquete en el que estamos trabajando).

```
String s" "hola";
System.out.print("hola");
```

String y **System** son clase de java.lang, se **pueden usar directamente**

```
package taller2.estructuras;
public class Vector {
  //código JAVA
}
```

Recomendación: usar como primera parte del **nombre del paquete el nombre invertido del dominio de Internet** y así evitar conflicto de nombres. Usar **minúscula para nombres de paquetes** e **inicial mayúscula para nombres de clases**.

Ejemplo: `ar.edu.unlp.graficos`

Librería de componentes

Paquetes JAVA

Un paquete normalmente está formado por varios archivos **.class**.
Java se beneficia de la **estructura jerárquica de directorios del sistema operativo** y ubica todos los **.class** de un mismo paquete en un mismo directorio. De esta manera, se resuelve:

- el nombre único del paquete
- la búsqueda de los **.class** (que de otra forma estarían diseminados en el disco)

```
package ar.edu.unlp.utiles;
public class Vector {
    //código JAVA
}
```

\ar\edu\unlp\utiles\Vector.class

Cuando el "intérprete" JAVA ejecuta un programa y necesita localizar dinámicamente un archivo **.class**, por ej. cuando se crea un objeto o se accede a un miembro **static**, procede de la siguiente manera:

- Busca en los **directorios estándares del JRE**
- Busca en el directorio actual (paquete de la clase que se está ejecutando)
- Recupera la variable de entorno **CLASSPATH**, que contiene la lista de directorios usados como raíces para buscar los archivos **.class**. Comenzando en la raíz, el intérprete toma el nombre del paquete (de las sentencias **import**) y reemplaza cada **"."** por una barra **"\"** o **"/"** (según el SO) para generar un camino donde encontrar las clases a partir de las entradas del **CLASSPATH**.

Librería de componentes

Paquetes JAVA

Consideremos el dominio **unlp.edu.ar** invertido y obtenemos un nombre de dominio único y global: **ar.edu.unlp**. Si creamos una librería **utiles** con las clases **Vector** y **List**, tendríamos:

```
package ar.edu.unlp.utiles;
public class Vector {
   //código JAVA
}
```

```
package ar.edu.unlp.utiles;
public class List {
   //código JAVA
}
```

Supongamos que a ambos archivos los guardamos en el directorio **c:\tallerjava**

```
C:\tallerjava\ar\edu\unlp\utiles\Vector.class
C:\tallerjava\ar\edu\unlp\utiles\List.class
```

El **"intérprete" JAVA** comienza a **buscar el paquete ar.edu.unlp a** partir de alguna de las **entradas** indicadas en la variable de entorno **CLASSPATH**:

```
CLASSPATH=.;c:\tallerjava;c:\java\librerias
```

Esta variable puede contener muchas entradas separadas por ";"

Paquetes en JAVA
Formato JAR

Es posible agrupar archivos **.class** pertenecientes a uno o más paquetes en un único archivo con extensión **jar (Java ARchive)**. El formato **JAR** usa el formato **zip**. Los archivos JAR son multi-plataforma, es estándar. Es posible incluir además de archivos .class, archivos de imágenes y audio, recursos en general, etc.

El JSE tiene una herramienta para crear archivos JAR, desde la línea de comando, es el utilitario **jar**.

Por ejemplo: si se ejecuta el comando **jar** desde el directorio donde están los archivos **.class** podríamos ponerlo así:

```
c:\tallerjava\ar\edu\unlp\utiles\jar cf utiles.jar *.class
```

- En este caso, en el **CLASSPATH** se especifica el nombre del archivo jar:

```
CLASSPATH=.;c:\utiles.jar;c:\java\librerias
```

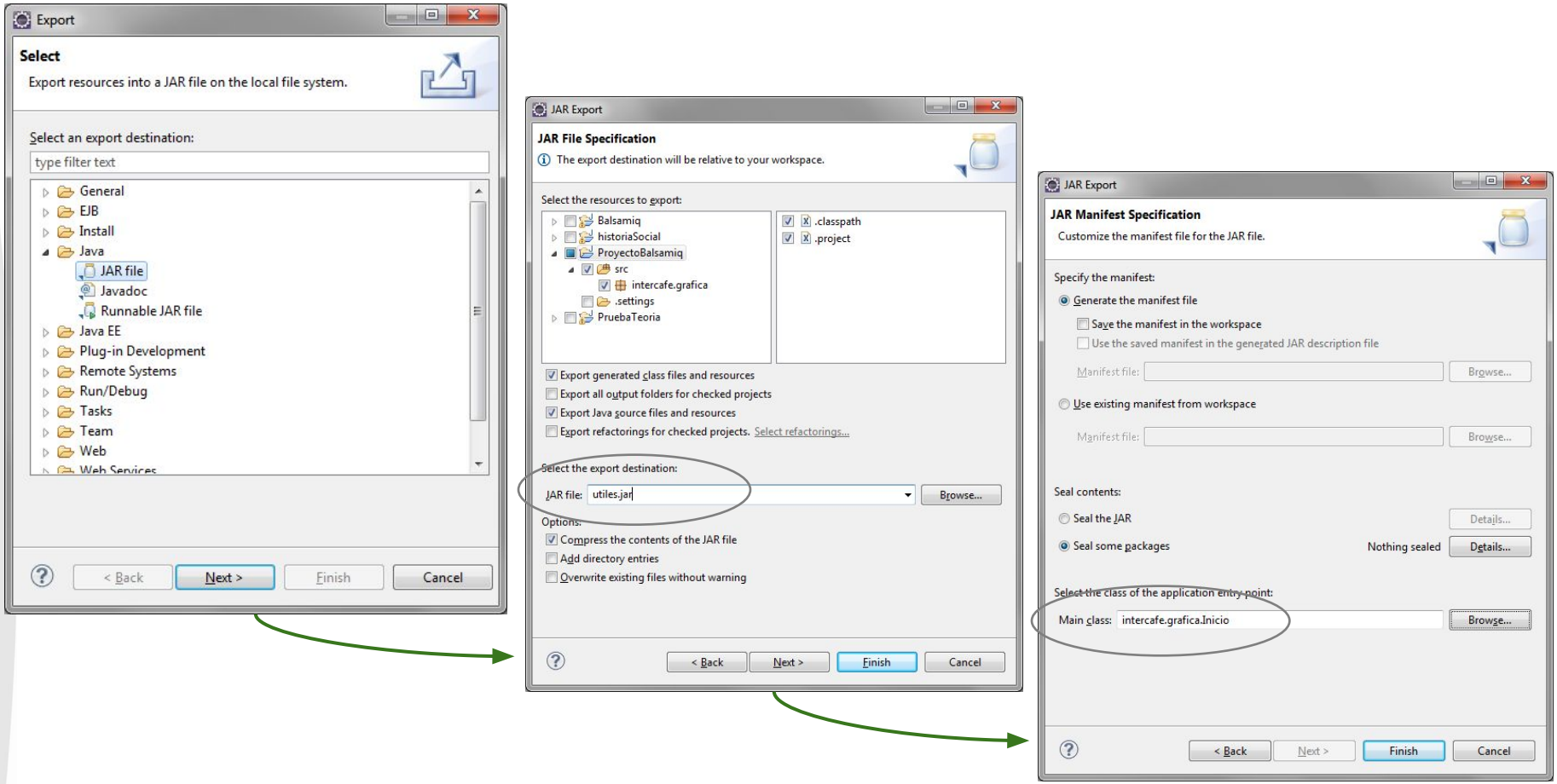
Los archivos jar pueden ubicarse en cualquier lugar del disco

- El "intérprete" JAVA se encarga de buscar, descomprimir, cargar e interpretar estos archivos.

Paquetes en JAVA

Formato JAR

El archivo JAR también puede construirse desde un proyecto Eclipse, con al opción **`export`**.



Paquetes en JAVA
El formato JAR

Los archivos JAR contienen todos los paquetes con sus archivos .class, los recursos de la aplicación y un archivo **MANIFEST.MF** ubicado en el camino META-INF/MANIFEST.MF, cuyo **propósito es indicar cómo se usa el archivo JAR**.

Las **aplicaciones de escritorio** a diferencia de las librerías de componentes o utilitarias, **requieren que el archivo MANIFEST.MF** contenga una **entrada con el nombre de la clase** que actuará como punto de entrada de la aplicación (la clase que contiene método main).

Para especificar la **clase "principal"**, el archivo MANIFEST.MF debe contener la **entrada Main-Class**.

```
Manifest-Version: 1.0
Created-By: 1.6.0_12 (Sun Microsystems Inc.)
Main-Class: capitulo4.paquetes.TestOut.class
```

PLAY!!!

kahoot.it