

IMPLEMENTACIÓN DE LA ENCAPSULACIÓN

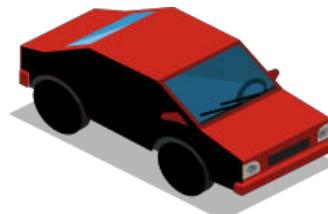


OBJETOS: ESTRUCTURA Y COMPORTAMIENTO

- ▶ En general, todos los objetos tienen una estructura (como están conformado) y un comportamiento (realizan una serie de operaciones).

ESTRUCTURA

- Plástico
- 4 ruedas
- 1 volante
- ...

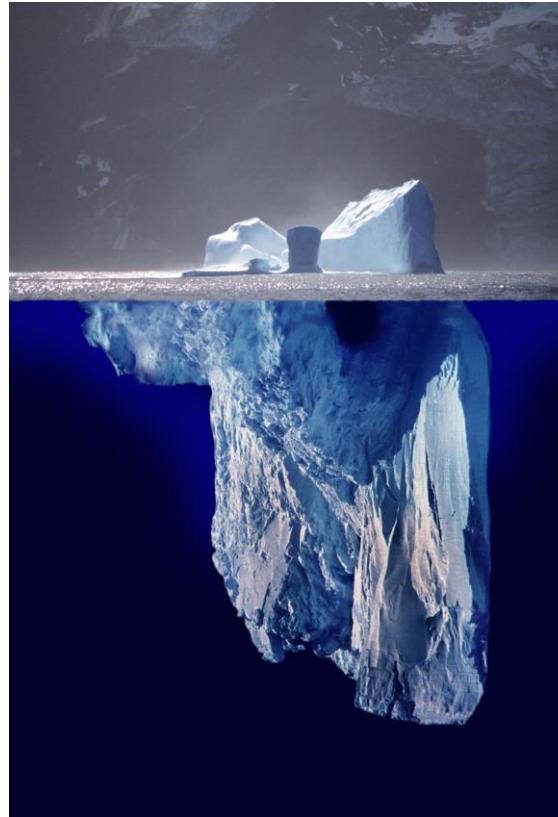


COMPORTAMIENTO

- Mover adelante
- Mover atrás
- ...

ENCAPSULACIÓN

- ▶ Los objetos conocen solamente su estructura, no la de los demás.



ENCAPSULACIÓN

- ▶ El trato entre objetos se realiza a través de los métodos.
- ▶ Normalmente, los atributos de un objeto se deben consultar o editar a través de métodos.



DEFINICIÓN E IMPLEMENTACIÓN DE UNA CLASE

```
<modificador> class NombreDeLaClase {  
  
    //propiedades  
    int propiedad1;  
    String propiedad2;  
    float propiedad3;  
    //...  
  
    //metodos  
    void metodo1() {  
        //...  
    }  
  
    //...  
}
```

MODIFICADORES DE ACCESO

- ▶ Nos permiten indicar quien puede hacer uso de una clase, o de sus atributos y métodos.
- ▶ *public*: cualquiera
- ▶ *private*: solo la propia clase
- ▶ *protected*: la propia clase y sus derivados
- ▶ Por defecto: las clases cercanas (que estén en el mismo paquete).

BEST PRACTICES

- ▶ La mayoría de las clases que se crean son públicas.
- ▶ Cada fichero .java tendrá solamente una clase pública, con el mismo nombre del fichero.

```
public class MiClase {  
  
    //propiedades  
    //...  
  
    //metodos  
    //...  
}
```



MiClase.java

BEST PRACTICES (II)

- ▶ La mayoría de los atributos de una clase serán privados.
- ▶ Solamente algunas constantes, o casos muy particulares, tendrán otra modificador de acceso.

```
public class MiClase {  
  
    //propiedades  
    private int numero;  
    private String nombre;  
  
    //metodos  
    //...  
}
```



MiClase.java

BEST PRACTICES (III)

- ▶ Si una clase tiene atributos, seguramente tenga métodos públicos.
- ▶ Los métodos privados son interesantes para cálculos auxiliares o parciales (solo se pueden invocar desde la propia clase).

```
public class MiClase {  
  
    //propiedades  
    private int numero;  
    private String nombre;  
  
    //metodos  
    public int getNumero() { ... }  
}
```



MiClase.java

TIPOS DE CLASES

- ▶ Java solo tiene “una forma” de crear clases, a través de **class**.

Podemos diferenciar las clases según su cometido:

- ▶ Modelo
- ▶ Servicios
- ▶ Auxiliares
- ▶ *Main*
- ▶ Test
- ▶ ...

TIPOS DE CLASES

- ▶ **Modelo:** representan objetos o hechos de la naturaleza: un coche, un asiento contable, los datos meteorológicos de un día. Suelen tener atributos, *getters* y *setters*, *equals*, *hashCode*, *toString*, ...
- ▶ **Servicios:** implementan la lógica de negocio. Suelen tener algunos atributos, pero sobre todo métodos públicos y privados.

TIPOS DE CLASES

- ▶ **Auxiliares:** sirven para realizar operaciones auxiliares de cálculo o transformación de datos.
Mayoritariamente, sus métodos son estáticos.
- ▶ **Main:** son el punto de entrada de la aplicación. La mayoría de las ocasiones, solo tienen este método, y si tienen más, suelen ser estáticos.

TIPOS DE CLASES

- ▶ **Test:** clases orientadas a realizar pruebas de nuestra aplicación. En Java, suelen ser test unitarios con JUnit.

HERENCIA y COMPOSICIÓN DE CLASES

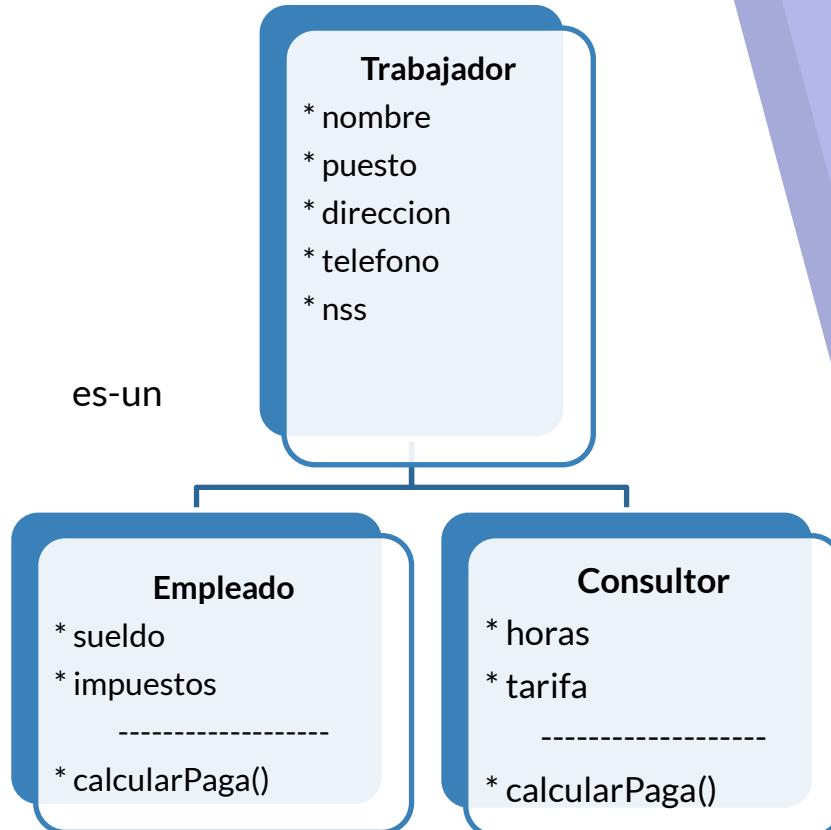


1.

HERENCIA DE CLASES

HERENCIA

- ▶ Una clase que extiende a otra hereda sus atributos y sus métodos (no constructores).
- ▶ Puede añadir atributos y métodos nuevos.



MODIFICADOR PROTECTED

- ▶ Si usamos protected en la clase base, tendremos acceso directo a los atributos.
- ▶ En otro caso, tendremos que acceder vía *getters/setters*.
- ▶ ¡OJO! Los constructores no se heredan aunque sean públicos.

2.

HERENCIA DE INTERFACES

HERENCIA DE INTERFACES

- ▶ También podemos establecer relaciones jerárquicas entre interfaces.
- ▶ Nos regimos por las mismas reglas que en el caso de las clases.

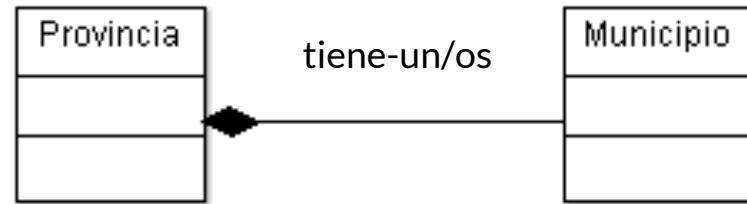
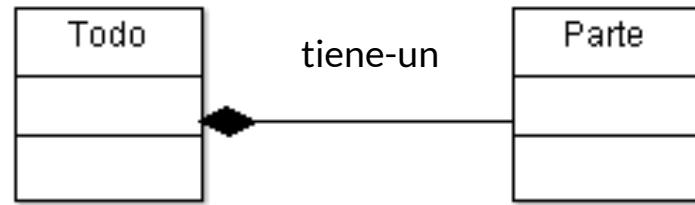
3.

COMPOSICIÓN DE CLASES

ASOCIACIONES ENTRE CLASES

- ▶ Normalmente, cuando representamos la estructura de un *sistema*, está formado por muchas clases.
- ▶ En este caso, no solamente importan las clases en sí, sino las *asociaciones*.
- ▶ Una de ellas es la **composición**.
- ▶ En UML, se representan de una forma especial.

ASOCIACIÓN DE COMPOSICIÓN



COMPOSICIÓN DE CLASES

- ▶ Dentro de la clase *Todo* tendremos una referencia a la clase *Parte*
- ▶ También es posible que la *multiplicidad* nos indique que debemos tener una colección (*Provincia* y *Municipio*).
- ▶ Normalmente hay dependencia de existencia entre la parte y el todo.

HERENCIA y COMPOSICIÓN DE CLASES

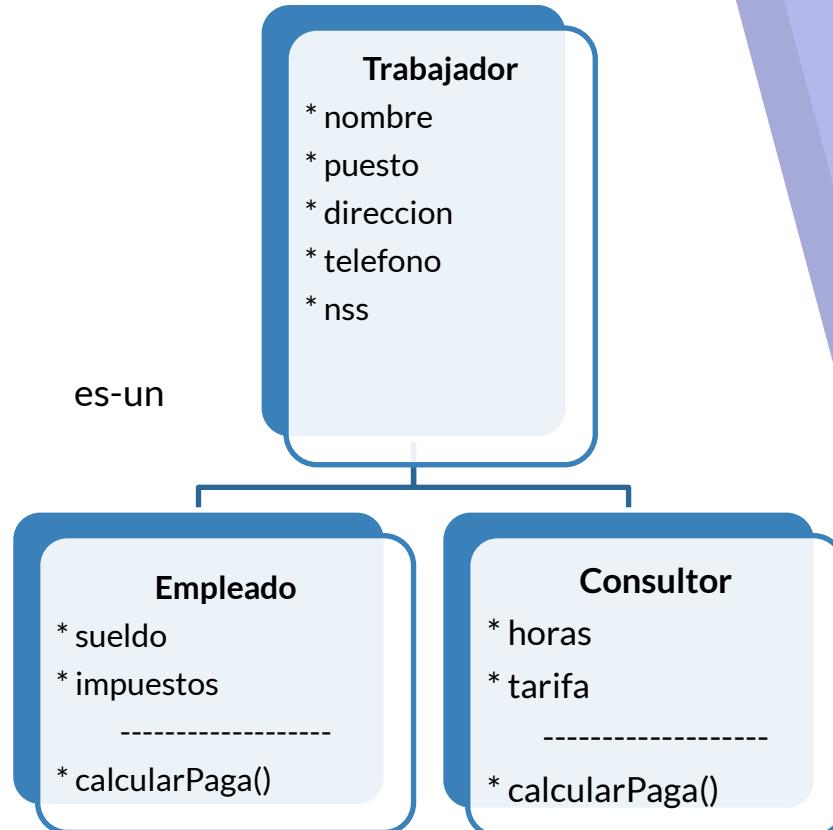


1.

HERENCIA DE CLASES

HERENCIA

- ▶ Una clase que extiende a otra hereda sus atributos y sus métodos (no constructores).
- ▶ Puede añadir atributos y métodos nuevos.



MODIFICADOR PROTECTED

- ▶ Si usamos protected en la clase base, tendremos acceso directo a los atributos.
- ▶ En otro caso, tendremos que acceder vía *getters/setters*.
- ▶ ¡OJO! Los constructores no se heredan aunque sean públicos.

2.

HERENCIA DE INTERFACES

HERENCIA DE INTERFACES

- ▶ También podemos establecer relaciones jerárquicas entre interfaces.
- ▶ Nos regimos por las mismas reglas que en el caso de las clases.

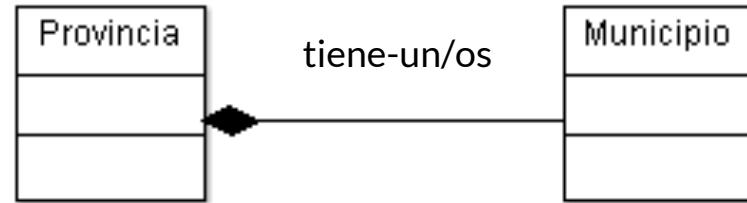
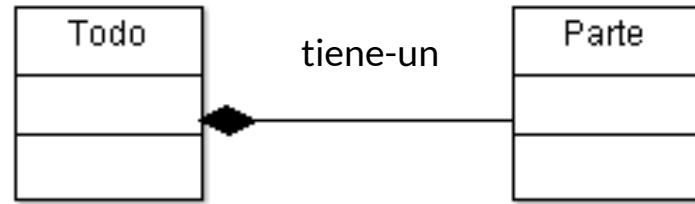
3.

COMPOSICIÓN DE CLASES

ASOCIACIONES ENTRE CLASES

- ▶ Normalmente, cuando representamos la estructura de un *sistema*, está formado por muchas clases.
- ▶ En este caso, no solamente importan las clases en sí, sino las *asociaciones*.
- ▶ Una de ellas es la **composición**.
- ▶ En UML, se representan de una forma especial.

ASOCIACIÓN DE COMPOSICIÓN



COMPOSICIÓN DE CLASES

- ▶ Dentro de la clase *Todo* tendremos una referencia a la clase *Parte*
- ▶ También es posible que la *multiplicidad* nos indique que debemos tener una colección (*Provincia* y *Municipio*).
- ▶ Normalmente hay dependencia de existencia entre la parte y el todo.



POLIMORFISMO

HERENCIA DE MÉTODOS

- ▶ Una subclase puede acceder a los métodos de su clase base (*public* y *protected*).
- ▶ También puede sobrescribir el comportamiento del mismo.

REFERENCIAS Y SUBCLASES

- ▶ Una subclase puede ser accedida a través de una referencia de una superclase.
- ▶ Esto es muy útil, sobre todo, para usar como atributos de métodos.

```
Rectangulo r = new Cuadrado();
```

OCULTACIÓN DE MÉTODOS

- ▶ Si una subclase añade un método con mismo nombre y firma que otro de la clase base, oculta a este.
- ▶ ¿Qué sucede en caso de el uso de referencias de clase base, pero instanciación de objetos derivados?

POLIMORFISMO

- ▶ Java escoge, en tiempo de ejecución, el tipo de objeto.
- ▶ Si ese tipo ha ocultado un método de la superclase, llama a la *concreción*.
- ▶ En otro caso, llama al método de la clase base.

POLIMORFISMO CON INTERFACES

- ▶ Java también hace uso de polimorfismo con la herencia de interfaces y las clases que lo implementan.

```
ClaseQueImplementaInterfaz c1 = new ClaseQueImplementaInterfaz();
c1.saludar("Hola Mundo");
```

```
Hija c2 = new ClaseQueImplementaInterfaz();
c2.saludar("Hola Mundo, otra vez");
```

```
Base c3 = new ClaseQueImplementaInterfaz();
c3.saludar("Hola Mundo, por tercera vez");
```

**EQUALS,
HASHCODE Y
TOSTRING**



HERENCIA DE OBJECT

- ▶ Todo objeto, de forma explícita o implícita, hereda de Object.
- ▶ Es la clase base de cualquier otra en Java.
- ▶ Tiene algunos métodos: equals, hashCode y toString.

COMPARACIÓN DE OBJETOS

- ▶ Con tipos primitivos, hemos usado el operador `==`.
- ▶ ¿Qué sucede con los objetos?
- ▶ Primero tenemos que definir cuando dos instancias de un objeto son iguales o diferentes.

Persona
nombre : String
apellidos : String
fechaNac : LocalDate

EQUALS

- ▶ El método **equals** nos permite devolver un boolean indicando si un objeto es igual a otro.
- ▶ Nuestro IDE lo autogenera.
- ▶ A la espera de una versión más compacta (Java SE 7).

<https://docs.oracle.com/javase/8/docs/api/java/util/Objects.html>

HASHCODE

- ▶ Devuelve un *número* asociado a la clase.
- ▶ Sirve como posición de memoria en hexadecimal.
- ▶ Por definición, si dos objetos son iguales (*equals*), su *hash code* también debe serlo.
- ▶ Si sobrescribimos el método *equals*, también tenemos que sobrescribir *hashCode* para que se cumpla esa propiedad.

TOSTRING

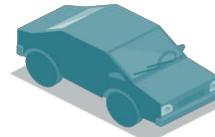
- ▶ Devuelve una representación en String del objeto.
- ▶ Por defecto, devuelve el tipo (la clase) y su hashCode.
- ▶ Lo podemos sobrescribir para que represente los valores.
- ▶ Dos objetos iguales deben tener la misma representación.

USO DE STATIC



ATRIBUTOS DE OBJETO Y DE CLASE

- ▶ Los objetos son instancias de una clase.
- ▶ Cada objeto tiene una copia de los atributos.
- ▶ ¿Y si quisiéramos tener un **atributo común** a todos los objetos de una clase?
- ▶ **static**



ATRIBUTOS ESTÁTICOS

- ▶ Están asociados a la clase, no a una instancia de ella.
- ▶ Son llamados **atributos estáticos**.
- ▶ **Compartidos** para todas las instancias de esa clase.
- ▶ Pueden ser manipulados por cualquier instancia.
- ▶ También pueden ser manipulados sin crear instancias de esa clase.

MÉTODOS ESTÁTICOS

- ▶ Similares a las variables estáticas (static)
- ▶ Se pueden invocar sin crear una instancia de esa clase.
- ▶ **Clase.metodoEstatico(...);**
- ▶ Podemos acceder una variable estática, necesitamos un método estático.
- ▶ Clases con métodos auxiliares (como por ejemplo, *java.util.Arrays*).

CONSTANTES

- ▶ Se suelen definir como estáticas.
- ▶ static final ...

```
static final double PI = 3.141592653589793;
```

- ▶ No se puede modificar su valor (error)
- ▶ Nombre en mayúsculas, separando palabras con guiones bajos.

CLASES ESTÁTICAS

- ▶ Válido para clases internas (las estudiaremos en profundidad más adelante).
- ▶ Nos permiten agrupar código.
- ▶ Para crear una instancia de la clase interna, no necesitamos una de la clase externa.

**CLASES
SINGLETON Y
CLASES
INMUTABLES**

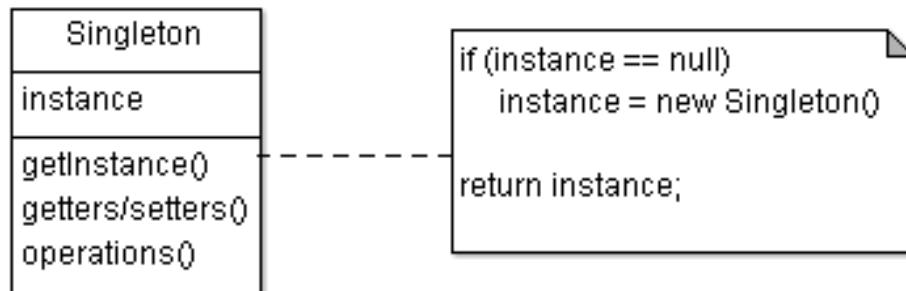


1.

SINGLETON

PATRÓN DE DISEÑO SINGLETON

- ▶ Patrón de diseño (GoF)
- ▶ Para aquellas clases de las que no queremos tener más de una instancia (por ejemplo, los servicios).



2. INMUTABLES

OBJETOS INMUTABLES

- ▶ Objetos cuyo estado no puede ser modificado una vez se haya inicializado.
- ▶ No son constantes (estas se definen en tiempo de compilación, y los inmutables en tiempo de ejecución).
- ▶ Un ejemplo de inmutable es **String**.

ESTRATEGIA PARA HACER OBJETOS INMUTABLES

- ▶ Definir las propiedades como **final** y **private**.
- ▶ No añadir métodos *setter*.
- ▶ Declarar la clase misma como **final**.

CLASES Y MÉTODOS ABSTRACTOS



ABSTRACT

- ▶ Palabra reservada
- ▶ Se puede usar a nivel de método y de clase.
- ▶ Sirve para indicar la obligación de implementar un método o extender una clase completa.

CLASES ABSTRACT

- ▶ Clase definida como **abstract**.
- ▶ No se pueden crear instancias de la misma.
- ▶ Puede tener métodos con implementación y atributos.

```
public abstract class AbstractaSencilla {  
  
    public void saluda() {  
        System.out.println("Hola mundo!!!");  
    }  
  
}
```

MÉTODOS ABSTRACT

- ▶ Deben estar en una clase definida como **abstract**.
- ▶ Definen la firma del método, pero sin implementación.
- ▶ Sus subclases se comprometen a implementarlo.
- ▶ Si no lo hacen, también deben ser abstractas.
- ▶ Pueden convivir con métodos normales.

```
public abstract class AbstractaConMetodos {  
  
    public abstract void saludo(String s);  
  
    public void saludar() {  
        System.out.println("Hola mundo!!!!");  
    }  
}
```

CLASES **ABSTRACT** QUE IMPLEMENTAN UNA INTERFAZ

- ▶ Una clase que implementa una interfaz tiene obligación de implementar todos sus métodos.
- ▶ Sin embargo, una clase **abstract** puede dejar métodos sin implementación, obligando a quienes la extiendan a hacerlo.

CÓDIGO QUE USA FINAL



MODIFICADOR FINAL

Se puede utilizar en diferentes contextos

- ▶ Clases final
- ▶ Métodos final
- ▶ Variables final

En todos los casos, indica que de una u otra forma, el ámbito sobre el que aplica no podrá ser modificado.

CLASES FINAL

- ▶ Son clases que no se pueden extender.
- ▶ En una jerarquía de herencia, son el último *nodo*.
- ▶ Se pueden instanciar y tratar con normalidad.

MÉTODOS FINAL

- ▶ Se definen en clases susceptibles de ser extendidas.
- ▶ Nos permiten indicar que un método no se va a poder sobrescribir.

VARIABLES FINAL

- ▶ Si van asociadas a tipos primitivos, son valores contantes (usualmente **final** + **static**).
- ▶ Si es una referencia al objeto, podemos modificar el estado del objeto; pero la referencia no puede apuntar a otro objeto diferente.
- ▶ Idem en el caso de un array.

CLASES INTERNAZ, LOCALES Y ANÓNIMAS



1.

CLASES ANIDADAS

CLASES DENTRO DE OTRAS CLASES

- ▶ Java permite definir clases dentro de otras clases.
- ▶ A estas clases se le llaman *anidadas*.
- ▶ Pueden ser de dos tipos, *estáticas* o *no estáticas*.
- ▶ No se trata de composición de clases, sino anidamiento.
- ▶ Pueden acceder a los atributos de la clase que le envuelve.

RAZONES PARA EL USO DE CLASES ANIDADAS

- ▶ Agrupamiento lógico de clases que se utilizan en un solo lugar. Mayor cohesión.
- ▶ Aumento de la encapsulación.
- ▶ Código más legible y fácil de mantener.

2.

CLASES INTERNAS

CLASES INTERNAS

- ▶ Se llaman así a las clases anidadas no estáticas.
- ▶ Solo pueden existir en el marco de una instancia de la clase externa.
- ▶ Pueden acceder a sus miembros (de la clase externa).

SHADOWING EN CLASES INTERNAS

- ▶ Si definimos una variable miembro en la clase interna, con el mismo nombre otra de la clase externa, la interna oculta a la externa.
- ▶ A esto se le llama *shadowing*.

3.

CLASES LOCALES

CLASES LOCALES

- ▶ Clases que se definen dentro de un bloque
(normalmente el cuerpo de un método)
- ▶ Afinan la cohesión del código a este nivel.

4.

CLASES ANÓNIMAS

CLASES ANÓNIMAS

- ▶ Permiten definir e instanciar una clase a la vez.
- ▶ Son como clases locales sin nombre.
- ▶ Sirven para ser usadas *una vez*.
- ▶ Las podemos definir a partir de otra clase o de una interfaz.
- ▶ Podemos crearlas en el cuerpo de un método, de una clase, o como argumento de un método.



USO DE ENUMERACIONES

TIPO ENUMERADO

- ▶ Tipo de dato especial
- ▶ Indica que una variable tendrá como valor uno de entre un conjunto cerrado.
- ▶ Por ejemplo Direccion (Norte, Sur, Este, Oeste).

```
public enum Direccion {  
    NORTE, SUR, ESTE, OESTE  
}
```

```
public enum Dia{  
    LUNES, MARTES,  
    MIERCOLES, JUEVES,  
    VIERNES, SABADO,  
    DOMINGO  
}
```

TIPO ENUMERADO

- ▶ Son más potentes que en otros lenguajes.
- ▶ Para Java son tipos de clases.
- ▶ Pueden incluir métodos y otros atributos.
- ▶ El compilador añade métodos especiales (*values*).
- ▶ Podemos pensar en que tenemos un conjunto cerrado de instancias de una clase.

CREACIÓN DE UNA CLASE GENÉRICA



CLASE GENÉRICA

- ▶ Se trata de una clase parametrizada sobre uno o más tipos.

```
public class Box {  
    private Object object;  
  
    public void set(Object object) {  
        this.object = object;  
    }  
  
    public Object get() {  
        return object;  
    }  
}
```



Este tipo de código es propenso a producir errores

CLASE GENÉRICA A PARTIR DE JAVA SE 5

```
public class Box<T> {  
    private T object;  
  
    public void set(T object) {  
        this.object = object;  
    }  
  
    public T get() {  
        return object;  
    }  
}
```

Se asigna el tipo en tiempo de compilación

CLASE GENÉRICA A PARTIR DE JAVA SE 5

```
public class Par<T, S> {  
    private T obj1;  
    private S obj2;  
  
    //Resto de la clase  
}
```

CONVENCIÓN SOBRE EL NOMBRE DE PARÁMETROS

Los nombres de tipos de parámetros más usados son:

- ▶ **E** (*element, elemento*)
- ▶ **K** (*key, clave*)
- ▶ **N** (*number, número*)
- ▶ **T** (*type, tipo*)
- ▶ **V** (*value, valor*)
- ▶ **S, U, V, ...** (2° , 3° , 4° , ... tipo)

INSTANCIACIÓN Y OPERADOR DIAMOND

- ▶ Para instanciar un objeto genérico, tenemos que indicar los tipos *dos veces*.

```
Par<String, String> pareja2 =  
    new Par<String, String>("Hola", "Mundo");
```

- ▶ Este estilo es muy *verboso*.
- ▶ Desde Java SE 7 tenemos el operador <> (*diamond*).

```
Par<String, String> pareja2 =  
    new Par<>("Hola", "Mundo");
```

GENÉRICOS CON TIPOS CERRADOS

- ▶ Podemos indicar que el tipo parametrizado sea uno en particular (o sus derivados).

```
public class NumericBox<T extends Number> {  
  
    private T object;  
  
    //resto de la clase  
}
```

GENÉRICOS CON TIPOS CERRADOS

- ▶ Podemos indicar más de un tipo
- ▶ Solo uno de ellos puede ser una clase.
- ▶ El resto deben ser interfaces
- ▶ La clase a extender debe ser la primera de la lista.

```
public class A {  
    //resto de la clase  
}  
  
public interface B {  
    //resto de la interfaz  
}
```

```
public class StrangeBox  
<T extends A & B> {  
    //resto de la clase  
}
```

GENÉRICOS CON TIPOS COMODÍN

- ▶ Nos permiten *relajar* el tipo concreto de una clase genérica a un subtipo.
- ▶ Útil con colecciones.

```
public static double sumOfList(List<? extends Number> list)
{
    double s = 0.0;
    for (Number n : list)
        s += n.doubleValue();
    return s;
}
```

CREACIÓN Y USO DE LIST, SET Y MAP



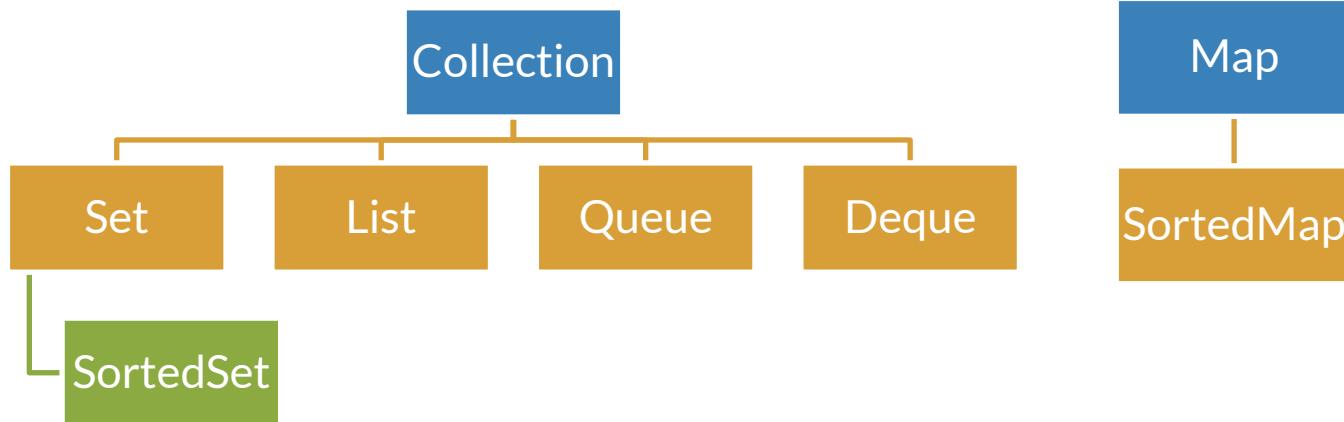
API COLLECTIONS

Desde Java SE 2 se ofrece el tratamiento de colecciones. Actualmente tiene

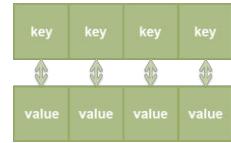
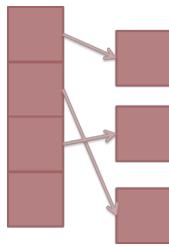
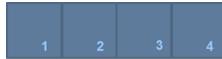
- ▶ **Interfaces:** tipos de datos
- ▶ **Implementaciones:** concreciones de los diferentes interfaces.
- ▶ **Algoritmos:** para realizar operaciones como ordenación, búsqueda, ...

Todas las colecciones están definidas como genéricas.

TIPOS DE COLECCIONES



TIPOS DE COLECCIONES



List

- Lineal
- Posibilidad de orden.
- Con repetidos

Set

- No soporta duplicados.
- Posibilidad de orden de elementos

Map

- Estructura clave, valor.
- Posibilidad de orden de elementos

1.

COLECCIONES LINEALES (LIST)

INTERFAZ LIST

- ▶ Los elementos tienen posición
- ▶ Permite duplicados
- ▶ También permite búsqueda e iteraciones
- ▶ Las implementaciones más conocidas son **ArrayList** y **LinkedList**.
- ▶ Si no sabemos cual escoger, utilizaremos siempre **ArrayList**.

CONSTRUCCIÓN DE UN LIST

A partir de Java 1.5

- ▶ Inclusión de los genéricos
- ▶ Permiten parametrizar el tipo

```
List<String> cars = new ArrayList<String>();
```

A partir de Java 1.7

- ▶ Operador *diamond*
- ▶ Nos ahorra indicar dos veces el tipo

```
List<String> cars = new ArrayList<>();
```

OPERACIONES CON LIST

Nombre	Uso
add	Añade un elemento al final la lista
addAll	Añade todos los elementos de la colección pasada como argumento
clear	Elimina todos los elementos de la lista.
contains	Comprueba si un elemento está o no en la lista
get	Devuelve el elemento de la posición especificada de la lista
isEmpty	Verifica si la lista está vacía
remove	Elimina un elemento de la lista
size	Devuelve el número de elementos de la lista
toArray	Devuelve la lista como un array

2.

COLECCIONES SIN REPETIDOS (SET)

INTERFAZ SET

- ▶ No puede contener repetidos.
- ▶ Propone tres implementaciones: **HashSet**, **TreeSet** y **LinkedHastSet**.
- ▶ **HashSet** es la más eficiente, pero no nos asegura nada sobre el orden.
- ▶ **TreeSet** utiliza un árbol Red-Black, ordena según el valor.
- ▶ **LinkedHashSet** es un HashSet ordenado por orden de inserción.

OPERACIONES CON SET

Nombre	Uso
add	Añade un elemento al conjunto, si aun no está contenido
addAll	Añade todos los elementos de la colección pasada como argumento si es que aun no están presentes.
clear	Elimina todos los elementos del conjunto.
contains	Comprueba si un elemento está o no en el conjunto
isEmpty	Verifica si el conjunto está vacío
remove	Elimina un elemento del conjunto
size	Devuelve el número de elementos de la lista
toArray	Devuelve la lista como un array

3.

COLECCIONES CALVE, VALOR

INTERFAZ MAP

- ▶ No es un subtipo de Collection (List y Set sí que lo son).
- ▶ Cada elemento tiene estructura clave, valor.
- ▶ La clave sirve para acceder directamente al valor.
- ▶ Las implementaciones son ***HashMap***, ***TreeMap*** y ***LinkedHashMap***. Las consideraciones son análogas a Set.

OPERACIONES CON MAP

Nombre	Uso
clear	Elimina todos los elementos del <i>diccionario</i> .
containsKey	Comprueba si una clave está presente en el <i>diccionario</i> .
containsValue	Comprueba si un valor está presente en el <i>diccionario</i> .
get	Devuelve el valor asociado a una clave.
isEmpty	Verifica si el conjunto está vacío
keySet	Devuelve un Set con todas las claves.
put	Permite insertar un par clave, valor
remove	Elimina un elemento del conjunto
size	Devuelve el número de elementos de la lista
values	Devuelve un Collection con los valores

INTERFACES COMPARABLE Y COMPARATOR



OPERACIONES CON OBJETOS

- ▶ Muchas operaciones entre objetos nos obligan a compararlos: buscar, ordenar, ...
- ▶ Los tipos primitivos y algunas clases ya implementan su orden (natural, lexicográfico).
- ▶ Para nuestras clases (*modelo*) tenemos que especificar el orden con el que las vamos a tratar.

1.

COMPARABLE

COMPARABLE

- ▶ Se trata de un interfaz sencillo

```
public interface Comparable<T> {  
    public int compareTo(T o);  
}
```

- ▶ Recibe un objeto del mismo tipo.
- ▶ Devuelve 0 si son iguales, un valor negativo si es menor, y uno positivo si es mayor.
- ▶ Nos sirve para indicar el **orden principal** de una clase.

2. COMPARATOR

COMPARATOR

- ▶ Se trata de un interfaz sencillo

```
public interface Comparator<T> {  
    public int compare(T o1, T o2);  
}
```

- ▶ Recibe dos argumentos
- ▶ Devuelve 0 si son iguales, un valor negativo si es menor, y uno positivo si es mayor.
- ▶ Nos sirve para indicar un orden puntual, diferente al **orden principal** de una clase.

INTERFACES FUNCIONALES



INTERFACES

- ▶ Contrato que compromete a una clase a implementar una serie de métodos.
- ▶ Se pueden utilizar como referencias a la hora de crear objetos.

```
List<String> lista = new ArrayList<>();
```

- ▶ Desde Java SE 8, pueden incluir implementación (static y default).

INTERFACES

INTERFAZ FUNCIONAL

- ▶ Interfaz que solo tiene un método abstracto.
- ▶ Puede tener uno o varios métodos por defecto y/o estáticos
- ▶ Puede tener varios métodos abstractos, siempre que *todos menos uno* sobrescriban un método público de la clase *Object*.
- ▶ **Usualmente, los implementamos con una clase anónima.**
- ▶ Muchos interfaces conocidos son funcionales.

INTERFAZ FUNCIONAL

- ▶ Java SE 8 también incorpora la anotación `@FunctionalInterface` que comprueba en tiempo de compilación si se cumplen con las condiciones anteriores.

INTERFACES FUNCIONALES Y EXPRESIONES LAMBDA

- ▶ Altamente relacionados
- ▶ De alguna forma, allí donde se espera una instancia de una clase que implemente una interfaz funcional, podremos usar una expresión lambda.

```
Collections.sort(lista, (str1, str2)-> str1.length()-str2.length());
```

HERENCIA y COMPOSICIÓN DE CLASES



1.

HERENCIA DE CLASES

HERENCIA

- ▶ Una clase que extiende a otra hereda sus atributos y sus métodos (no constructores).
- ▶ Puede añadir atributos y métodos nuevos.



MODIFICADOR PROTECTED

- ▶ Si usamos protected en la clase base, tendremos acceso directo a los atributos.
- ▶ En otro caso, tendremos que acceder vía *getters/setters*.
- ▶ ¡OJO! Los constructores no se heredan aunque sean públicos.

2.

HERENCIA DE INTERFACES

HERENCIA DE INTERFACES

- ▶ También podemos establecer relaciones jerárquicas entre interfaces.
- ▶ Nos regimos por las mismas reglas que en el caso de las clases.

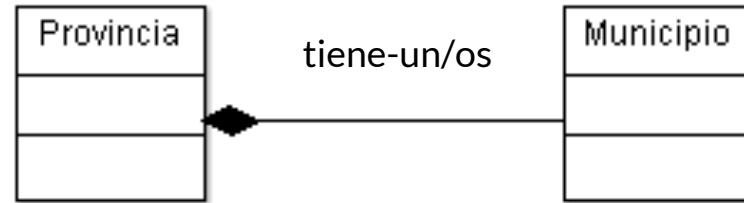
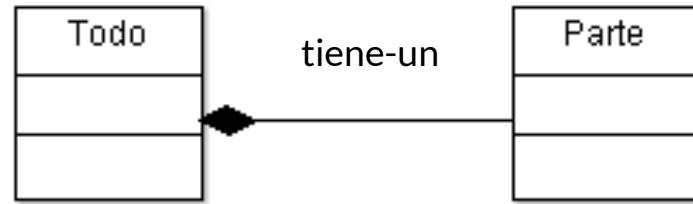
3.

COMPOSICIÓN DE CLASES

ASOCIACIONES ENTRE CLASES

- ▶ Normalmente, cuando representamos la estructura de un *sistema*, está formado por muchas clases.
- ▶ En este caso, no solamente importan las clases en sí, sino las *asociaciones*.
- ▶ Una de ellas es la **composición**.
- ▶ En UML, se representan de una forma especial.

ASOCIACIÓN DE COMPOSICIÓN



COMPOSICIÓN DE CLASES

- ▶ Dentro de la clase *Todo* tendremos una referencia a la clase *Parte*
- ▶ También es posible que la *multiplicidad* nos indique que debemos tener una colección (*Provincia* y *Municipio*).
- ▶ Normalmente hay dependencia de existencia entre la parte y el todo.



POLIMORFISMO

HERENCIA DE MÉTODOS

- ▶ Una subclase puede acceder a los métodos de su clase base (*public* y *protected*).
- ▶ También puede sobrescribir el comportamiento del mismo.

REFERENCIAS Y SUBCLASES

- ▶ Una subclase puede ser accedida a través de una referencia de una superclase.
- ▶ Esto es muy útil, sobre todo, para usar como atributos de métodos.

```
Rectangulo r = new Cuadrado();
```

OCULTACIÓN DE MÉTODOS

- ▶ Si una subclase añade un método con mismo nombre y firma que otro de la clase base, oculta a este.
- ▶ ¿Qué sucede en caso de el uso de referencias de clase base, pero instanciación de objetos derivados?

POLIMORFISMO

- ▶ Java escoge, en tiempo de ejecución, el tipo de objeto.
- ▶ Si ese tipo ha ocultado un método de la superclase, llama a la *concreción*.
- ▶ En otro caso, llama al método de la clase base.

POLIMORFISMO CON INTERFACES

- ▶ Java también hace uso de polimorfismo con la herencia de interfaces y las clases que lo implementan.

```
ClaseQueImplementaInterfaz c1 = new ClaseQueImplementaInterfaz();
c1.saludar("Hola Mundo");
```

```
Hija c2 = new ClaseQueImplementaInterfaz();
c2.saludar("Hola Mundo, otra vez");
```

```
Base c3 = new ClaseQueImplementaInterfaz();
c3.saludar("Hola Mundo, por tercera vez");
```

**EQUALS,
HASHCODE Y
TOSTRING**



HERENCIA DE OBJECT

- ▶ Todo objeto, de forma explícita o implícita, hereda de Object.
- ▶ Es la clase base de cualquier otra en Java.
- ▶ Tiene algunos métodos: equals, hashCode y toString.

COMPARACIÓN DE OBJETOS

- ▶ Con tipos primitivos, hemos usado el operador `==`.
- ▶ ¿Qué sucede con los objetos?
- ▶ Primero tenemos que definir cuando dos instancias de un objeto son iguales o diferentes.

Persona
nombre : String
apellidos : String
fechaNac : LocalDate

EQUALS

- ▶ El método **equals** nos permite devolver un boolean indicando si un objeto es igual a otro.
- ▶ Nuestro IDE lo autogenera.
- ▶ A la espera de una versión más compacta (Java SE 7).

<https://docs.oracle.com/javase/8/docs/api/java/util/Objects.html>

HASHCODE

- ▶ Devuelve un *número* asociado a la clase.
- ▶ Sirve como posición de memoria en hexadecimal.
- ▶ Por definición, si dos objetos son iguales (*equals*), su *hash code* también debe serlo.
- ▶ Si sobrescribimos el método *equals*, también tenemos que sobrescribir *hashCode* para que se cumpla esa propiedad.

TOSTRING

- ▶ Devuelve una representación en String del objeto.
- ▶ Por defecto, devuelve el tipo (la clase) y su hashCode.
- ▶ Lo podemos sobrescribir para que represente los valores.
- ▶ Dos objetos iguales deben tener la misma representación.

USO DE STATIC



ATRIBUTOS DE OBJETO Y DE CLASE

- ▶ Los objetos son instancias de una clase.
- ▶ Cada objeto tiene una copia de los atributos.
- ▶ ¿Y si quisiéramos tener un **atributo común** a todos los objetos de una clase?
- ▶ **static**



ATRIBUTOS ESTÁTICOS

- ▶ Están asociados a la clase, no a una instancia de ella.
- ▶ Son llamados **atributos estáticos**.
- ▶ **Compartidos** para todas las instancias de esa clase.
- ▶ Pueden ser manipulados por cualquier instancia.
- ▶ También pueden ser manipulados sin crear instancias de esa clase.

MÉTODOS ESTÁTICOS

- ▶ Similares a las variables estáticas (static)
- ▶ Se pueden invocar sin crear una instancia de esa clase.
- ▶ **Clase.metodoEstatico(...);**
- ▶ Podemos acceder una variable estática, necesitamos un método estático.
- ▶ Clases con métodos auxiliares (como por ejemplo, *java.util.Arrays*).

CONSTANTES

- ▶ Se suelen definir como estáticas.
- ▶ static final ...

```
static final double PI = 3.141592653589793;
```

- ▶ No se puede modificar su valor (error)
- ▶ Nombre en mayúsculas, separando palabras con guiones bajos.

CLASES ESTÁTICAS

- ▶ Válido para clases internas (las estudiaremos en profundidad más adelante).
- ▶ Nos permiten agrupar código.
- ▶ Para crear una instancia de la clase interna, no necesitamos una de la clase externa.

**CLASES
SINGLETON Y
CLASES
INMUTABLES**

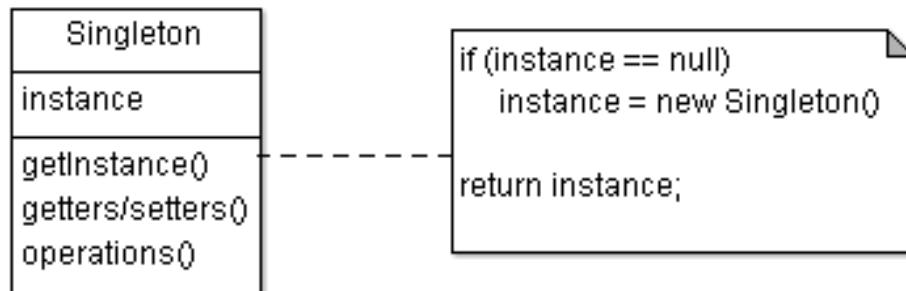


1.

SINGLETON

PATRÓN DE DISEÑO SINGLETON

- ▶ Patrón de diseño (GoF)
- ▶ Para aquellas clases de las que no queremos tener más de una instancia (por ejemplo, los servicios).



2. INMUTABLES

OBJETOS INMUTABLES

- ▶ Objetos cuyo estado no puede ser modificado una vez se haya inicializado.
- ▶ No son constantes (estas se definen en tiempo de compilación, y los inmutables en tiempo de ejecución).
- ▶ Un ejemplo de inmutable es **String**.

ESTRATEGIA PARA HACER OBJETOS INMUTABLES

- ▶ Definir las propiedades como **final** y **private**.
- ▶ No añadir métodos *setter*.
- ▶ Declarar la clase misma como **final**.

CLASES Y MÉTODOS ABSTRACTOS



ABSTRACT

- ▶ Palabra reservada
- ▶ Se puede usar a nivel de método y de clase.
- ▶ Sirve para indicar la obligación de implementar un método o extender una clase completa.

CLASES ABSTRACT

- ▶ Clase definida como **abstract**.
- ▶ No se pueden crear instancias de la misma.
- ▶ Puede tener métodos con implementación y atributos.

```
public abstract class AbstractaSencilla {  
  
    public void saluda() {  
        System.out.println("Hola mundo!!!");  
    }  
  
}
```

MÉTODOS ABSTRACT

- ▶ Deben estar en una clase definida como **abstract**.
- ▶ Definen la firma del método, pero sin implementación.
- ▶ Sus subclases se comprometen a implementarlo.
- ▶ Si no lo hacen, también deben ser abstractas.
- ▶ Pueden convivir con métodos normales.

```
public abstract class AbstractaConMetodos {  
  
    public abstract void saludo(String s);  
  
    public void saludar() {  
        System.out.println("Hola mundo!!!!");  
    }  
}
```

CLASES **ABSTRACT** QUE IMPLEMENTAN UNA INTERFAZ

- ▶ Una clase que implementa una interfaz tiene obligación de implementar todos sus métodos.
- ▶ Sin embargo, una clase **abstract** puede dejar métodos sin implementación, obligando a quienes la extiendan a hacerlo.

CÓDIGO QUE USA FINAL



MODIFICADOR FINAL

Se puede utilizar en diferentes contextos

- ▶ Clases final
- ▶ Métodos final
- ▶ Variables final

En todos los casos, indica que de una u otra forma, el ámbito sobre el que aplica no podrá ser modificado.

CLASES FINAL

- ▶ Son clases que no se pueden extender.
- ▶ En una jerarquía de herencia, son el último *nodo*.
- ▶ Se pueden instanciar y tratar con normalidad.

MÉTODOS FINAL

- ▶ Se definen en clases susceptibles de ser extendidas.
- ▶ Nos permiten indicar que un método no se va a poder sobrescribir.

VARIABLES FINAL

- ▶ Si van asociadas a tipos primitivos, son valores contantes (usualmente **final** + **static**).
- ▶ Si es una referencia al objeto, podemos modificar el estado del objeto; pero la referencia no puede apuntar a otro objeto diferente.
- ▶ Idem en el caso de un array.

CLASES INTERNAZ, LOCALES Y ANÓNIMAS



1.

CLASES ANIDADAS

CLASES DENTRO DE OTRAS CLASES

- ▶ Java permite definir clases dentro de otras clases.
- ▶ A estas clases se le llaman *anidadas*.
- ▶ Pueden ser de dos tipos, *estáticas* o *no estáticas*.
- ▶ No se trata de composición de clases, sino anidamiento.
- ▶ Pueden acceder a los atributos de la clase que le envuelve.

RAZONES PARA EL USO DE CLASES ANIDADAS

- ▶ Agrupamiento lógico de clases que se utilizan en un solo lugar. Mayor cohesión.
- ▶ Aumento de la encapsulación.
- ▶ Código más legible y fácil de mantener.

2.

CLASES INTERNAS

CLASES INTERNAS

- ▶ Se llaman así a las clases anidadas no estáticas.
- ▶ Solo pueden existir en el marco de una instancia de la clase externa.
- ▶ Pueden acceder a sus miembros (de la clase externa).

SHADOWING EN CLASES INTERNAS

- ▶ Si definimos una variable miembro en la clase interna, con el mismo nombre otra de la clase externa, la interna oculta a la externa.
- ▶ A esto se le llama *shadowing*.

3.

CLASES LOCALES

CLASES LOCALES

- ▶ Clases que se definen dentro de un bloque
(normalmente el cuerpo de un método)
- ▶ Afinan la cohesión del código a este nivel.

4.

CLASES ANÓNIMAS

CLASES ANÓNIMAS

- ▶ Permiten definir e instanciar una clase a la vez.
- ▶ Son como clases locales sin nombre.
- ▶ Sirven para ser usadas *una vez*.
- ▶ Las podemos definir a partir de otra clase o de una interfaz.
- ▶ Podemos crearlas en el cuerpo de un método, de una clase, o como argumento de un método.



USO DE ENUMERACIONES

TIPO ENUMERADO

- ▶ Tipo de dato especial
- ▶ Indica que una variable tendrá como valor uno de entre un conjunto cerrado.
- ▶ Por ejemplo Direccion (Norte, Sur, Este, Oeste).

```
public enum Direccion {  
    NORTE, SUR, ESTE, OESTE  
}
```

```
public enum Dia{  
    LUNES, MARTES,  
    MIERCOLES, JUEVES,  
    VIERNES, SABADO,  
    DOMINGO  
}
```

TIPO ENUMERADO

- ▶ Son más potentes que en otros lenguajes.
- ▶ Para Java son tipos de clases.
- ▶ Pueden incluir métodos y otros atributos.
- ▶ El compilador añade métodos especiales (*values*).
- ▶ Podemos pensar en que tenemos un conjunto cerrado de instancias de una clase.

CREACIÓN DE UNA CLASE GENÉRICA



CLASE GENÉRICA

- ▶ Se trata de una clase parametrizada sobre uno o más tipos.

```
public class Box {  
    private Object object;  
  
    public void set(Object object) {  
        this.object = object;  
    }  
  
    public Object get() {  
        return object;  
    }  
}
```



Este tipo de código es propenso a producir errores

CLASE GENÉRICA A PARTIR DE JAVA SE 5

```
public class Box<T> {  
    private T object;  
  
    public void set(T object) {  
        this.object = object;  
    }  
  
    public T get() {  
        return object;  
    }  
}
```

Se asigna el tipo en tiempo de compilación

CLASE GENÉRICA A PARTIR DE JAVA SE 5

```
public class Par<T, S> {  
    private T obj1;  
    private S obj2;  
  
    //Resto de la clase  
}
```

CONVENCIÓN SOBRE EL NOMBRE DE PARÁMETROS

Los nombres de tipos de parámetros más usados son:

- ▶ **E** (*element, elemento*)
- ▶ **K** (*key, clave*)
- ▶ **N** (*number, número*)
- ▶ **T** (*type, tipo*)
- ▶ **V** (*value, valor*)
- ▶ **S, U, V, ...** (2° , 3° , 4° , ... tipo)

INSTANCIACIÓN Y OPERADOR DIAMOND

- ▶ Para instanciar un objeto genérico, tenemos que indicar los tipos *dos veces*.

```
Par<String, String> pareja2 =  
    new Par<String, String>("Hola", "Mundo");
```

- ▶ Este estilo es muy *verboso*.
- ▶ Desde Java SE 7 tenemos el operador <> (*diamond*).

```
Par<String, String> pareja2 =  
    new Par<>("Hola", "Mundo");
```

GENÉRICOS CON TIPOS CERRADOS

- ▶ Podemos indicar que el tipo parametrizado sea uno en particular (o sus derivados).

```
public class NumericBox<T extends Number> {  
  
    private T object;  
  
    //resto de la clase  
}
```

GENÉRICOS CON TIPOS CERRADOS

- ▶ Podemos indicar más de un tipo
- ▶ Solo uno de ellos puede ser una clase.
- ▶ El resto deben ser interfaces
- ▶ La clase a extender debe ser la primera de la lista.

```
public class A {  
    //resto de la clase  
}  
  
public interface B {  
    //resto de la interfaz  
}
```

```
public class StrangeBox  
<T extends A & B> {  
    //resto de la clase  
}
```

GENÉRICOS CON TIPOS COMODÍN

- ▶ Nos permiten *relajar* el tipo concreto de una clase genérica a un subtipo.
- ▶ Útil con colecciones.

```
public static double sumOfList(List<? extends Number> list)
{
    double s = 0.0;
    for (Number n : list)
        s += n.doubleValue();
    return s;
}
```

CREACIÓN Y USO DE LIST, SET Y MAP



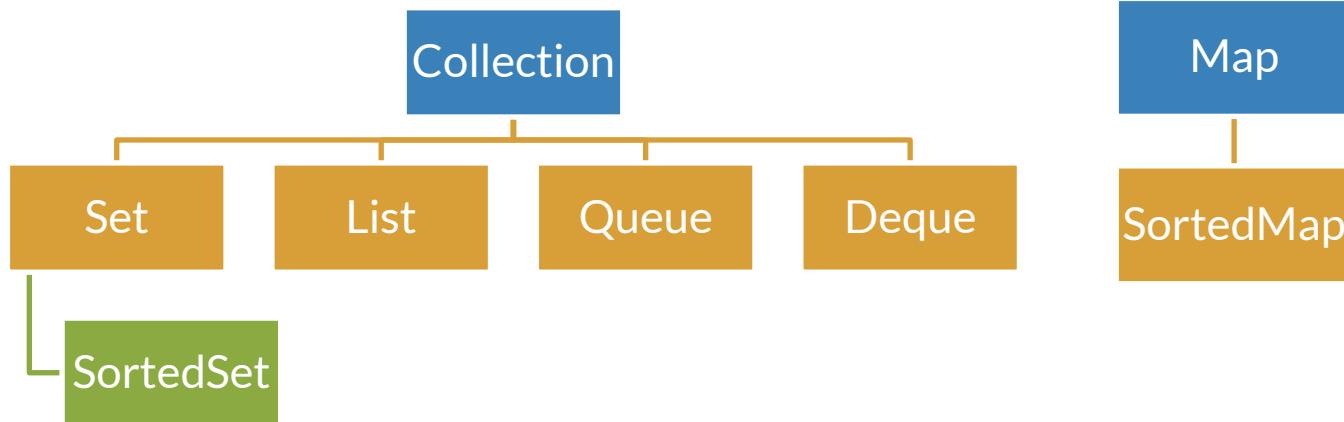
API COLLECTIONS

Desde Java SE 2 se ofrece el tratamiento de colecciones. Actualmente tiene

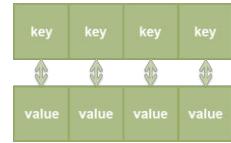
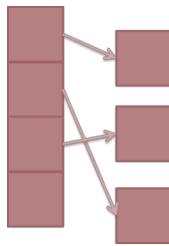
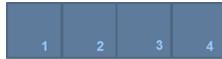
- ▶ **Interfaces:** tipos de datos
- ▶ **Implementaciones:** concreciones de los diferentes interfaces.
- ▶ **Algoritmos:** para realizar operaciones como ordenación, búsqueda, ...

Todas las colecciones están definidas como genéricas.

TIPOS DE COLECCIONES



TIPOS DE COLECCIONES



List

- Lineal
- Posibilidad de orden.
- Con repetidos

Set

- No soporta duplicados.
- Posibilidad de orden de elementos

Map

- Estructura clave, valor.
- Posibilidad de orden de elementos

1.

COLECCIONES LINEALES (LIST)

INTERFAZ LIST

- ▶ Los elementos tienen posición
- ▶ Permite duplicados
- ▶ También permite búsqueda e iteraciones
- ▶ Las implementaciones más conocidas son **ArrayList** y **LinkedList**.
- ▶ Si no sabemos cual escoger, utilizaremos siempre **ArrayList**.

CONSTRUCCIÓN DE UN LIST

A partir de Java 1.5

- ▶ Inclusión de los genéricos
- ▶ Permiten parametrizar el tipo

```
List<String> cars = new ArrayList<String>();
```

A partir de Java 1.7

- ▶ Operador *diamond*
- ▶ Nos ahorra indicar dos veces el tipo

```
List<String> cars = new ArrayList<>();
```

OPERACIONES CON LIST

Nombre	Uso
add	Añade un elemento al final la lista
addAll	Añade todos los elementos de la colección pasada como argumento
clear	Elimina todos los elementos de la lista.
contains	Comprueba si un elemento está o no en la lista
get	Devuelve el elemento de la posición especificada de la lista
isEmpty	Verifica si la lista está vacía
remove	Elimina un elemento de la lista
size	Devuelve el número de elementos de la lista
toArray	Devuelve la lista como un array

2.

COLECCIONES SIN REPETIDOS (SET)

INTERFAZ SET

- ▶ No puede contener repetidos.
- ▶ Propone tres implementaciones: **HashSet**, **TreeSet** y **LinkedHastSet**.
- ▶ **HashSet** es la más eficiente, pero no nos asegura nada sobre el orden.
- ▶ **TreeSet** utiliza un árbol Red-Black, ordena según el valor.
- ▶ **LinkedHashSet** es un HashSet ordenado por orden de inserción.

OPERACIONES CON SET

Nombre	Uso
add	Añade un elemento al conjunto, si aun no está contenido
addAll	Añade todos los elementos de la colección pasada como argumento si es que aun no están presentes.
clear	Elimina todos los elementos del conjunto.
contains	Comprueba si un elemento está o no en el conjunto
isEmpty	Verifica si el conjunto está vacío
remove	Elimina un elemento del conjunto
size	Devuelve el número de elementos de la lista
toArray	Devuelve la lista como un array

3.

COLECCIONES CALVE, VALOR

INTERFAZ MAP

- ▶ No es un subtipo de Collection (List y Set sí que lo son).
- ▶ Cada elemento tiene estructura clave, valor.
- ▶ La clave sirve para acceder directamente al valor.
- ▶ Las implementaciones son *HashMap*, *TreeMap* y *LinkedHashMap*. Las consideraciones son análogas a Set.

OPERACIONES CON MAP

Nombre	Uso
clear	Elimina todos los elementos del <i>diccionario</i> .
containsKey	Comprueba si una clave está presente en el <i>diccionario</i> .
containsValue	Comprueba si un valor está presente en el <i>diccionario</i> .
get	Devuelve el valor asociado a una clave.
isEmpty	Verifica si el conjunto está vacío
keySet	Devuelve un Set con todas las claves.
put	Permite insertar un par clave, valor
remove	Elimina un elemento del conjunto
size	Devuelve el número de elementos de la lista
values	Devuelve un Collection con los valores

INTERFACES COMPARABLE Y COMPARATOR



OPERACIONES CON OBJETOS

- ▶ Muchas operaciones entre objetos nos obligan a compararlos: buscar, ordenar, ...
- ▶ Los tipos primitivos y algunas clases ya implementan su orden (natural, lexicográfico).
- ▶ Para nuestras clases (*modelo*) tenemos que especificar el orden con el que las vamos a tratar.

1.

COMPARABLE

COMPARABLE

- ▶ Se trata de un interfaz sencillo

```
public interface Comparable<T> {  
    public int compareTo(T o);  
}
```

- ▶ Recibe un objeto del mismo tipo.
- ▶ Devuelve 0 si son iguales, un valor negativo si es menor, y uno positivo si es mayor.
- ▶ Nos sirve para indicar el **orden principal** de una clase.

2. COMPARATOR

COMPARATOR

- ▶ Se trata de un interfaz sencillo

```
public interface Comparator<T> {  
    public int compare(T o1, T o2);  
}
```

- ▶ Recibe dos argumentos
- ▶ Devuelve 0 si son iguales, un valor negativo si es menor, y uno positivo si es mayor.
- ▶ Nos sirve para indicar un orden puntual, diferente al **orden principal** de una clase.

INTERFACES FUNCIONALES



INTERFACES

- ▶ Contrato que compromete a una clase a implementar una serie de métodos.
- ▶ Se pueden utilizar como referencias a la hora de crear objetos.

```
List<String> lista = new ArrayList<>();
```

- ▶ Desde Java SE 8, pueden incluir implementación (static y default).

INTERFACES

INTERFAZ FUNCIONAL

- ▶ Interfaz que solo tiene un método abstracto.
- ▶ Puede tener uno o varios métodos por defecto y/o estáticos
- ▶ Puede tener varios métodos abstractos, siempre que *todos menos uno* sobrescriban un método público de la clase *Object*.
- ▶ **Usualmente, los implementamos con una clase anónima.**
- ▶ Muchos interfaces conocidos son funcionales.

INTERFAZ FUNCIONAL

- ▶ Java SE 8 también incorpora la anotación `@FunctionalInterface` que comprueba en tiempo de compilación si se cumplen con las condiciones anteriores.

INTERFACES FUNCIONALES Y EXPRESIONES LAMBDA

- ▶ Altamente relacionados
- ▶ De alguna forma, allí donde se espera una instancia de una clase que implemente una interfaz funcional, podremos usar una expresión lambda.

```
Collections.sort(lista, (str1, str2)-> str1.length()-str2.length());
```

USO DE INTERFACES COMO PREDICATE, CONSUMER, FUNCTION Y SUPPLIER



1.

PREDICATE

PREDICATE

- ▶ Método **boolean test(T t)**.
- ▶ Comprueba si se cumple o no una determinada condición.
- ▶ Muy utilizado con expresiones lambda para filtrar

```
listaPersonas
    .stream()
    .filter((p) -> p.getEdad() >= 35)
    .forEach(System.out::println);
```

PREDICATE

- ▶ Métodos útiles para construir predicados complejos (*and, or, negate...*)

```
Predicate<Persona> edad = (p) -> p.getEdad() >= 351;  
Predicate<Persona> nombre = (p) ->  
                    p.getApellidos().contains("e");  
Predicate<Persona> complejo = edad.or(nombre);
```

2. CONSUMER

CONSUMER

- ▶ Método **void accept(T t)**.
- ▶ Sirve para consumir objetos.
- ▶ El ejemplo más claro es imprimir
- ▶ Muy utilizado con expresiones lambda para imprimir.

```
listaPersonas
    .stream()
    .filter((p) -> p.getEdad() >= 351)
    .forEach(System.out::println);
```

CONSUMER

- ▶ Adicionalmente, tiene el método `andThen`, que permite componer *consumidores*, y encadenar así una secuencia de operaciones.

```
Consumer<Integer> consumer = i -> System.out.println(i);
Consumer<Integer> consumerWithAndThen =
    consumer.andThen(i -> System.out.println(
        "hemos imprimido el " + i + "));
```

3.

FUNCTION

FUNCTION

- ▶ Método *R apply(T t)*.
- ▶ Sirve para aplicar una transformación sobre un objeto.
- ▶ El ejemplo más claro es el mapeo de un objeto en otro.
- ▶ Muy utilizado con expresiones lambda para mapear.

```
lista
    .stream()
    .map((p) -> p.getNombre())
    .foreach(System.out::println);
```

FUNCTION

Adicionalmente, tiene otros métodos:

- ▶ *andThen*, que permite componer funciones.
- ▶ *compose*, que compone dos funciones, a la inversa de la anterior.
- ▶ *identity*, una función que siempre devuelve el argumento que recibe

4.

SUPPLIER

SUPPLIER

- ▶ Método **T get()**.
- ▶ No recibe ningún parámetro
- ▶ Sirve para obtener objetos.
- ▶ Su uso está menos extendido que las anteriores.
- ▶ Su sintaxis como expresión lambda sería

```
() -> something  
() -> { return something; }
```

SUPPLIER

Tiene interfaces especializados para tipos básicos

- ▶ *IntSupplier*
- ▶ *LongSupplier*
- ▶ *DoubleSupplier*
- ▶ *BooleanSupplier*



INTRODUCCIÓN AL API STREAM

API STREAM

- ▶ Una de las grandes novedades de Java SE 8, junto a las expresiones lambda.
- ▶ Permite realizar operaciones de filtro/mapeo/reducción sobre colecciones de datos.
- ▶ Puede trabajar de forma secuencia o paralela.
- ▶ Transparente al desarrollador.
- ▶ Combinación perfecta para las expresiones lambda.

STREAM

Es una secuencia de elementos que soporta operaciones para procesarlos

- ▶ Usando expresiones lambda
- ▶ Permitiendo el encadenamiento de operaciones (código legible y conciso).
- ▶ De forma secuencial o paralela.

Definido por la interfaz
java.util.stream.Stream<T>

CARACTERÍSTICAS DE UN STREAM

- ▶ Las operaciones intermedias retornan un *Stream* (encadenamiento).
- ▶ Las operaciones intermedias se encolan, y son invocadas al invocar una operación terminal.
- ▶ Solo se puede recorrer una vez.
- ▶ Iteración interna vs. externa: nos centramos en qué hacer con los datos, no en como recorrerlos.

SUBTIPOS BÁSICOS

- ▶ IntStream
- ▶ LongStream
- ▶ DoubleStream

FORMAS DE OBTENER UN STREAM

- ▶ *Stream.of(...)* ordenado y secuencial de los parámetros que se le pasan
- ▶ *Arrays.stream(T[])* secuencial a partir del array proporcionado. Si el array es de tipo básico, se devuelve un subtipo de Stream.
- ▶ *Stream.empty()* devuelve un stream secuencial y vacío.

FORMAS DE OBTENER UN STREAM

- ▶ `Stream.iterate(T, UnaryOperator<T>)` devuelve un stream infinito, ordenado, y secuencial, a partir de un valor y de aplicar una función a ese valor. Se puede limitar su tamaño con `limit(long)`.
- ▶ `Collection<T>.stream()`,
`Collection<T>.parallelStream()` devuelve un stream (secuencial o paralelo) a partir de una colección.

FORMAS DE OBTENER UN **STREAM**

- ▶ `Stream.generate(Supplier<T>)` retorna un stream infinito, secuencial y no ordenado a partir de un *Supplier*.

OPERACIONES INTERMEDIAS SOBRE UN **STREAM**

- ▶ Son operaciones que devuelven un stream.
- ▶ Nos permiten realizar diversas funciones (filtrado, transformación, ...)

OPERACIONES DE FILTRADO

- ▶ *filter(Predicate<T>)*: nos permite filtrar utilizando una condición.
- ▶ *limit(n)*: nos permite obtener los n primeros elementos.
- ▶ *skip(m)*: nos permite *obviar* los primeros m elementos.

OPERACIONES DE MAPEO

- ▶ `map(Function<T, R>)`: nos permite transformar los valores de un stream a través de una expresión lambda o instancia de `Function`.
- ▶ `mapToInt(...)`, `mapToDouble(...)`, `mapToLong(...)` nos permite transformar los valores en uno de estos tipos básicos, obteniendo un `IntStream`, `DoubleStream`, `LongStream`, respectivamente.

OPERACIONES TERMINALES SOBRE UN STREAM

Provoca que se ejecuten todas las operaciones terminales.

Varios tipos:

- ▶ Consumo de elementos: `forEach(...)`
- ▶ Obtener datos del stream
- ▶ Recolección de elementos para transformarlos en otro objeto, como una colección.

Los dos últimos tipos los estudiaremos en lecciones posteriores.

API STREAM: MÉTODOS DE BÚSQUEDA DE DATOS



MÉTODOS DE BÚSQUEDA

Operaciones terminales sobre un stream.

Nos permiten:

- ▶ Identificar si hay elementos que cumplan una condición.
- ▶ Obtener (si el stream contiene elementos) algunos elementos en particular.

MÉTODOS DE BÚSQUEDA

- ▶ *allMatch(Predicate<T>)* verifica si todos los elementos de un stream satisfacen un predicado.
- ▶ *anyMatch(Predicate<T>)* verifica si algún elemento de un stream satisface un predicado.
- ▶ *noneMatch(Predicate<T>)* opuesto de *allMatch(...)*

MÉTODOS DE BÚSQUEDA

- ▶ *findAny()* devuelve en un Optional<T> un elemento (cualquiera) del stream.
Recomendado en streams paralelos.
- ▶ *findFirst()* devuelve en un Optional<T> el primer elemento del stream. NO RECOMENDADO en streams paralelos.

MÉTODOS DE DATOS, CÁLCULO Y ORDENACIÓN



MÉTODOS DE DATOS Y CÁLCULO

Los streams nos ofrecen métodos terminales para hacer múltiples operaciones con los datos.

Vamos a trabajar con tres tipos:

- ▶ Reducción y resumen
- ▶ Agrupamiento
- ▶ Particionamiento

Los trataremos en próximas lecciones

MÉTODOS DE REDUCCIÓN

- ▶ *reduce(BinaryOperator<T>):Optional<T>*

Realiza la reducción del Stream usando una función asociativa. Devuelve un *Optional*

- ▶ *reduce(T, BinaryOperator<T>):T*

Realiza la reducción usando un valor inicial y una función asociativa. Se devuelve un valor como resultado.

MÉTODOS DE RESUMEN

- ▶ `count():long`

Devuelve el número de elementos del Stream.

- ▶ `min(Comparator<T>): Optional<T>,`

- ▶ `max(Comparator<T>): Optional<T>`

Devuelven el mínimo o máximo de un stream basándose en el comparador que reciben como parámetro.

`Comparator` ofrece algunos método estáticos útiles para este cometido.

MÉTODOS DE ORDENACIÓN

Son operaciones intermedias. Nos ofrecen un stream con los elementos ordenados (de alguna manera)

- ▶ `sorted()` el stream se ordena según el orden natural.
- ▶ `sorted(Comparator<T>)` el stream se ordena según el orden indicado por la instancia de Comparator.

USO DE MAP, FLATMAP Y COLLECTOR



1. MAP

MAP

- ▶ Una de las operaciones intermedias más usadas.
- ▶ Permite aplicar una transformación a una serie de objetos.
- ▶ Recibe como argumento un Function<T,R> para realizar la transformación.
- ▶ Se invoca sobre un Stream<T>, y retorna un Stream<R>

MAP

- ▶ Se pueden realizar transformaciones sucesivas

```
lista
    .stream()                      Stream<Persona>
    .map(Persona::getNombre)        Stream<String>
    .map(String::toUpperCase)       Stream<String>
    .forEach(System.out::println);
```

2.

FLATMAP

FLATMAP

- ▶ Los streams sobre colecciones de *un nivel* (como *List*) se pueden transformar (*map*) fácilmente.
 - ▶ ¿Qué sucede si tenemos una colección que incluye dentro otra?

Persona 1			Persona 2			Persona 3		
Viaje 1	Viaje 2	Viaje 3	Viaje 4	Viaje 5	Viaje 6	Viaje 7	Viaje 8	Viaje 9

FLATMAP

- ▶ En el viejo estilo for, anidaríamos dos bucles:

```
for (Persona p : lista)
    for (Viaje v : p.getViajes())
        System.out.println(v.getPais());
```

FLATMAP

- ▶ Podemos observar bien para darnos cuenta los tipos de retorno de los métodos intermedios:

```
// Intento sin flatMap
lista
    .stream()                               Stream<Persona>
    .map((Persona p) -> p.getViajes())     Stream<Stream<Viaje>>
    //...
    .forEach(System.out::println);
```

FLATMAP

- ▶ Necesitamos un método que unifique un $\text{Stream} < \text{Stream} < T > >$ en un solo $\text{Stream} < T >$:
- ▶ Esa es la funcionalidad de *flatMap*.

$\text{Stream} < \text{Stream} < \text{Viaje} > >$

Viaje 1	Viaje 2	Viaje 3	Viaje 4	Viaje 5	Viaje 6	Viaje 7	Viaje 8	Viaje 9
------------	------------	------------	------------	------------	------------	------------	------------	------------

$\text{Stream} < \text{Viaje} >$

Viaje 1	Viaje 2	Viaje 3	Viaje 4	Viaje 5	Viaje 6	Viaje 7	Viaje 8	Viaje 9
------------	------------	------------	------------	------------	------------	------------	------------	------------

FLATMAP

- ▶ Con flatMap podemos transformar los streams y unificarlos en uno:

```
lista
    .stream()
    .map((Persona p) -> p.getViajes())
    .flatMap(viajes -> viajes.stream())
    .map((Viaje v) -> v.getPais())
    .forEach(System.out::println);
```

FLATMAP

- ▶ También tenemos la versiones primitivas (flatMapToInt,...):

```
int[][] numeros = { {1, 2, 2, 3, 1, 4},  
                    {4, 2, 3, 3, 1, 1} };
```

Arrays

```
.stream(numeros)  
.flatMapToInt(x -> Arrays.stream(x))  
.map(IntUnaryOperator.identity())  
.distinct()  
.forEach(System.out::println);
```

3. COLLECTORS

COLLECTORS

- ▶ Hasta ahora, las operaciones realizadas con streams han acabado con una salida por consola.
- ▶ ¿Y si queremos transformar un stream (*immutable*) y guardar su resultado en una colección (*mutable*)? Operación *collect*.
- ▶ Java SE 8 introduce **Collectors**, con métodos estáticos muy usuales (y prácticos).

```
import static java.util.stream.Collectors.*;
```

COLLECTORS “BÁSICOS”

- ▶ Nos permite realizar algún tipo de operación y recolectar el valor en uno solo.
- ▶ Algunos se solapan con operaciones finales que ya hemos visto; existen para usarse junto con otros colectores.

COLLECTORS “BÁSICOS”

- ▶ *counting()*: cuenta el número de elementos.
- ▶ *minBy(...)*, *maxBy(...)*: obtiene el mínimo o máximo según un comparador.
- ▶ *summingInt*, *summingLong*, *summingDouble*: la suma de los elementos (según el tipo).
- ▶ *averagingInt*, *averagingLong*, *averagingDouble*: la media (según el tipo).
- ▶ *summarizingInt*, *summarizingLong*,
summarizingDouble: los valores anteriores, agrupados en un objeto (según el tipo).
- ▶ *joining*: unión de los elementos en una cadena.

COLLECTORS “GROUPING BY”

- ▶ Similar a la cláusula GROUP BY de SQL.
- ▶ Permiten agrupar los elementos de un stream por un determinado valor.
- ▶ Retorna un *Map* con los diferentes grupos, y los elementos de cada grupo.

```
Map<String, List<Empleado>> porDepartamento =  
    empleados  
        .stream()  
        .collect(groupingBy(Empleado::getDepartamento));
```

COLLECTORS “GROUPING BY”

- ▶ Se pueden usar uno de los colectores básicos, para realizar algún cálculo

```
Map<String, Long> porDepartamentoCantidad =  
    empleados  
        .stream()  
        .collect(groupingBy(Empleado::getDepartamento, counting()));
```

COLLECTORS “GROUPING BY”

- ▶ Se pueden crear varios niveles de agrupamiento:

COLLECTORS “PARTITIONING”

- ▶ Se pueden agrupar en dos conjuntos, según si cumplen la condición de un predicado.

```
Map<Boolean, List<Empleado>> salarioMayor0Igualque32000 =  
    empleados  
        .stream()  
        .collect(partitioningBy(e -> e.getSalario() >= 32000));
```

COLLECTORS “COLLECTION”

- ▶ Producen como resultado una de las colecciones ya conocidas: List y Set
- ▶ También puede producir colecciones de tipo key, value, como Map.



USO DE STREAMS Y FILTROS

FILTER

- ▶ Operación intermedia
- ▶ Nos permite eliminar a aquellos elementos del stream que no cumplen una determinada condición
- ▶ Condición como *Predicate*.
- ▶ Muy combinable con *findFirst*, *findAny*
- ▶ Combinable con el resto de métodos intermedios y terminales.

REFERENCIAS A MÉTODOS



REFERENCIAS A MÉTODOS

- ▶ Una forma de hacer el código aun más conciso.

```
public class Persona {  
//...  
    public static int  
        compararPorEdad(Persona a, Persona b) {  
            return a.fechaNacimiento  
                .compareTo(b.fechaNacimiento);  
    }  
}
```

REFERENCIAS A MÉTODOS

- ▶ Una forma de hacer el código aun más conciso.

```
personas.sort((Persona p1, Persona p2) -> {  
    return p1.getFechaNacimiento()  
        .compareTo(p2.getFechaNacimiento());  
});
```

```
personas.sort((p1, p2) -> p1.getFechaNacimiento()  
                .compareTo(p2.getFechaNacimiento()));  
  
personas.sort(Persona::compararPorEdad);
```

TIPOS DE REFERENCIAS

- ▶ **Clase::metodoEstatico:** referencia a un método estático.
- ▶ **objeto::metodoInstancia:** referencia a un método de instancia de un objeto concreto.
- ▶ **Tipo::nombreMetodo:** referencia a un método de instancia de un objeto *arbitrario* de un tipo en particular.
- ▶ **Clase::new:** referencia a un constructor.

FLUJOS DE SALIDA



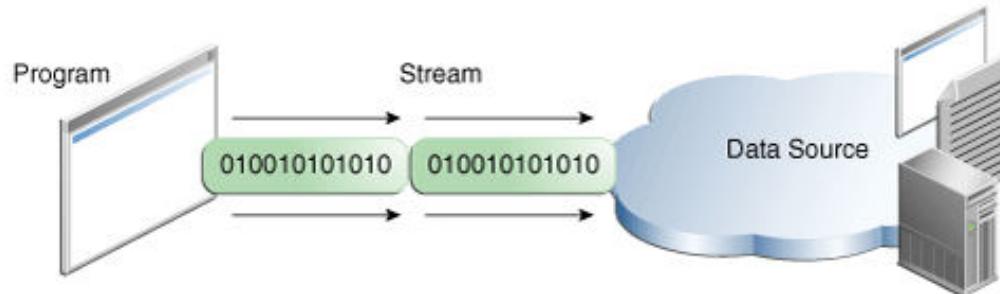
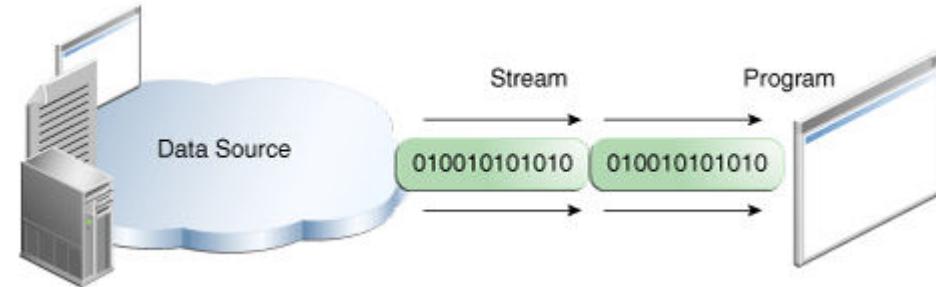
FLUJO

- ▶ Canal de comunicación de las operaciones de entrada/salida
- ▶ Literalmente, es un flujo (producción/consumo de información).

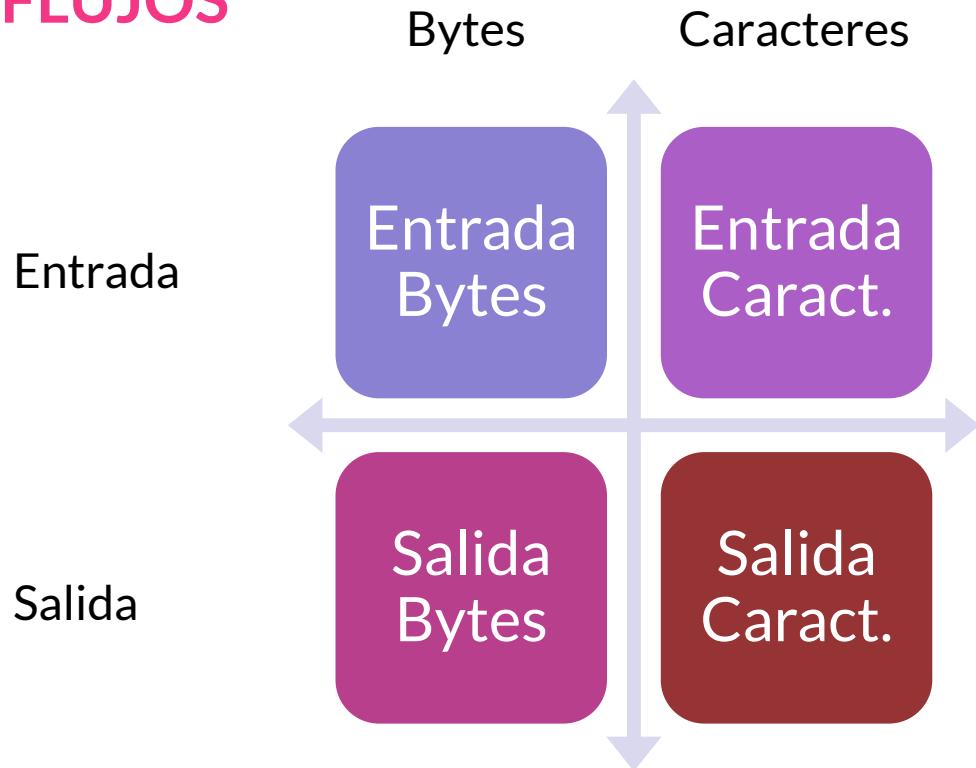
Nos da independencia:

- ▶ Entrada desde el teclado
- ▶ Salida hacia el monitor
- ▶ Lectura de un fichero
- ▶ Envío de datos por red
- ▶ ...

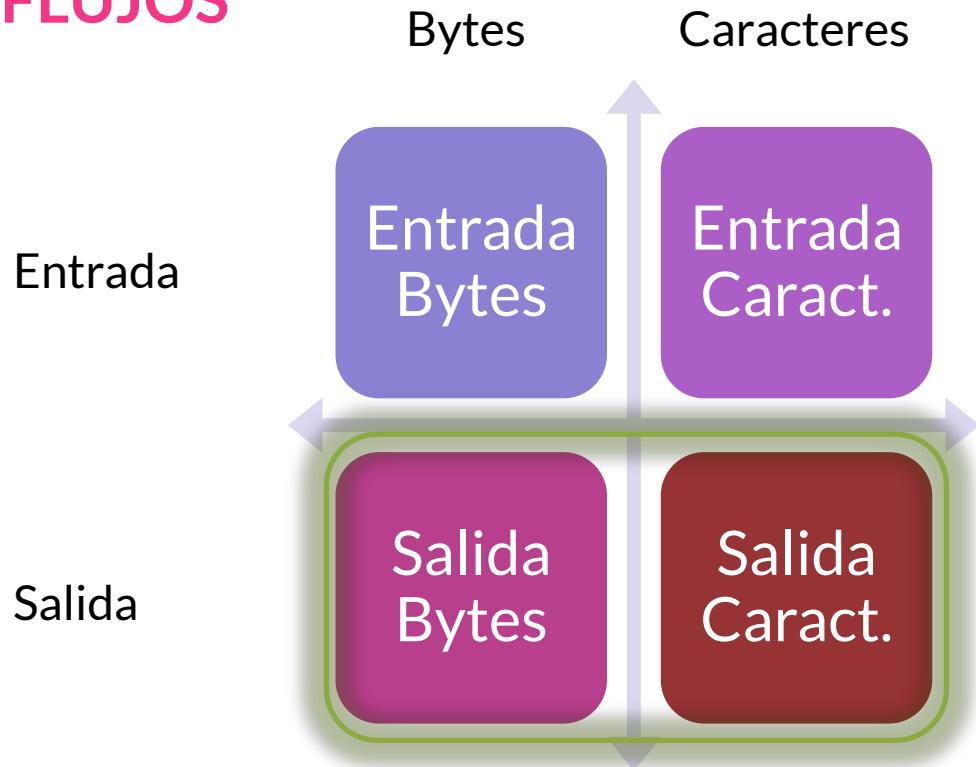
FLUJO



FLUJOS



FLUJOS



FLUJOS DE SALIDA

- ▶ Patrón básico de uso de flujos de salida:

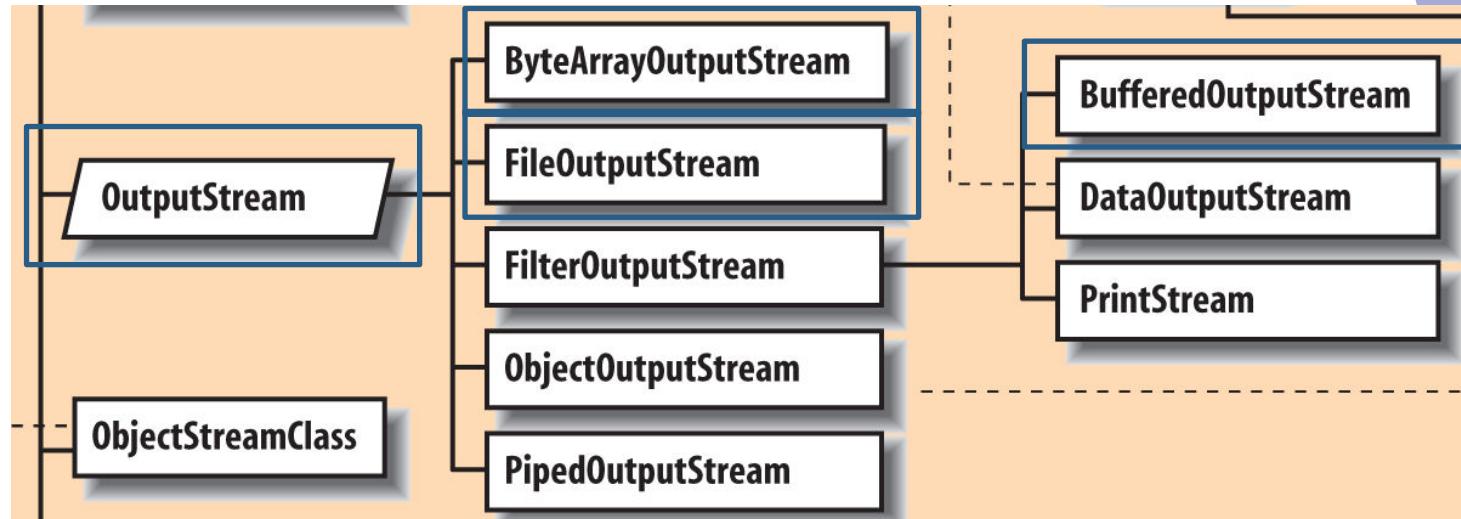
Abrir el flujo

Mientras hay datos que escribir

 Escribir datos en el flujo

Cerrar el flujo

FLUJOS DE SALIDA DE BYTES

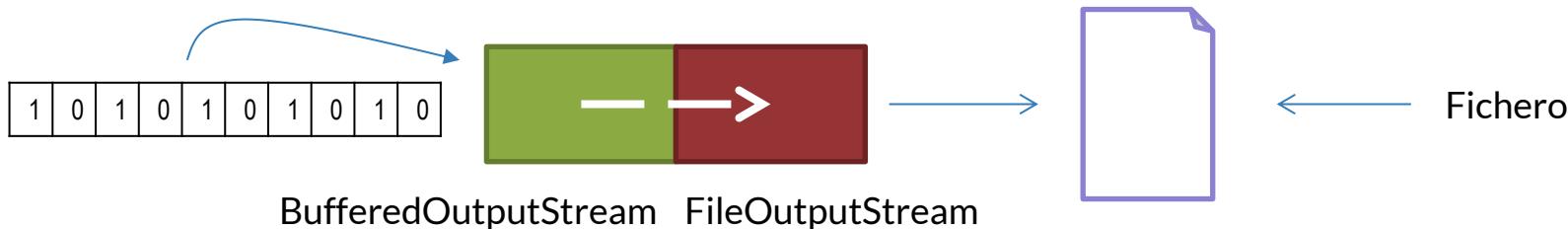


FLUJOS DE SALIDA DE BYTES

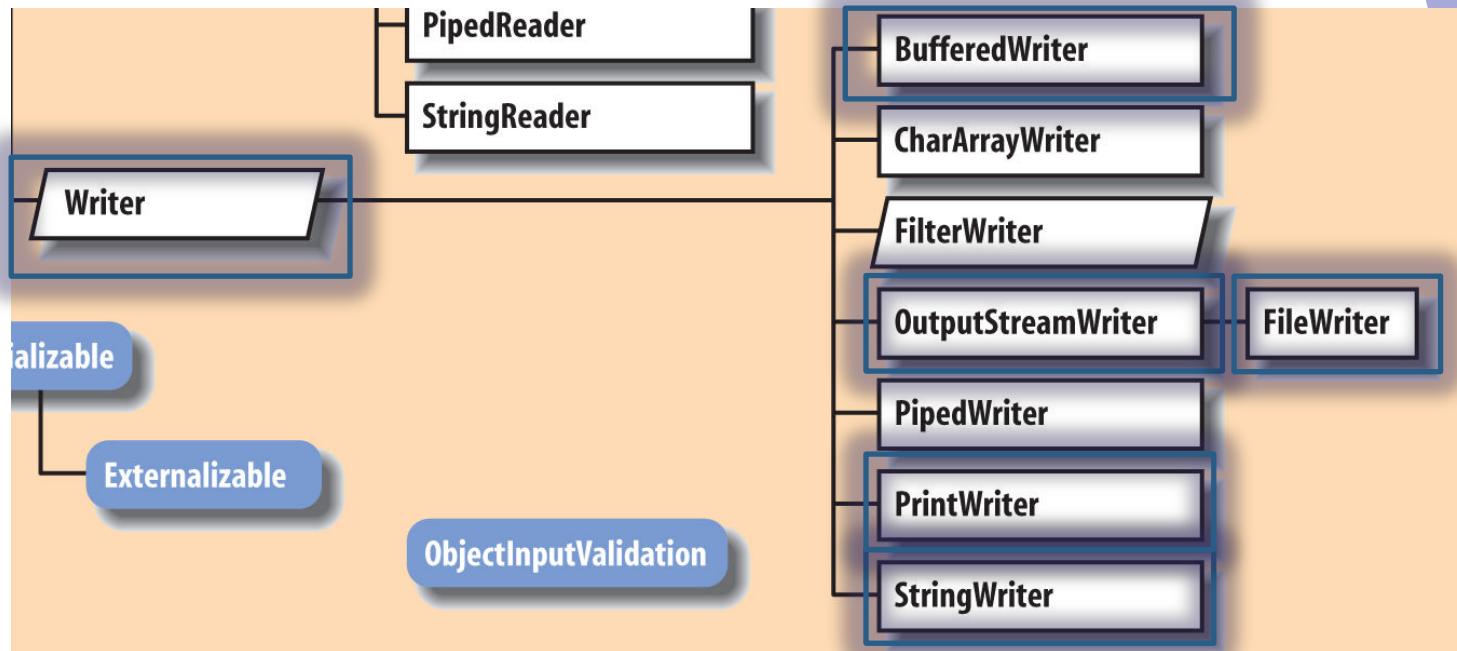
- ▶ **OutputStream**: clase abstracta, padre de la mayoría de los flujos de bytes.
- ▶ **FileOutputStream**: flujo que permite escribir en un fichero, byte a byte.
- ▶ **BufferedOutputStream**: flujo que permite escribir grupos (buffers) de bytes.
- ▶ **ByteArrayOutputStream**: flujo que permite escribir *en memoria*, obteniendo lo escrito en un array de bytes.

FLUJOS HACIA OTROS FLUJOS

- ▶ Solo *FileOutputStream* tiene un constructor que acepta una ruta (entre otras opciones).
- ▶ El resto reciben en sus constructores *OutputStream*. ¿Por qué?
- ▶ Podemos construir flujos que escriben en flujos (encadenados).



FLUJOS DE SALIDA DE CARACTERES



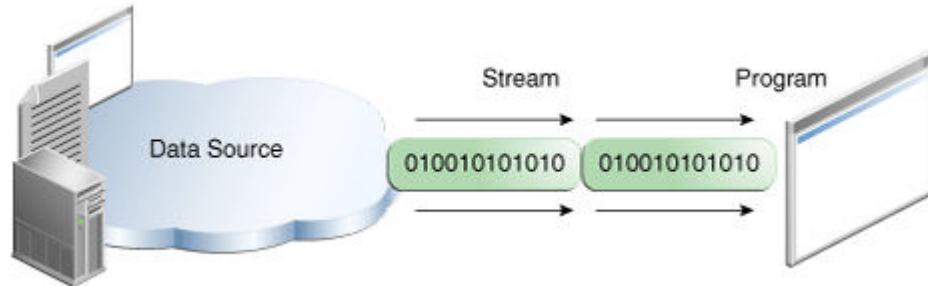
FLUJOS DE SALIDA DE CARACTERES

- ▶ **Writer**: clase abstracta, padre de la mayoría de los flujos de caracteres.
- ▶ **FileWriter**: flujo que permite escribir en un fichero, carácter a carácter.
- ▶ **BufferedWriter**: flujo que permite escribir líneas de texto.
- ▶ **StringWriter**: flujo que permite escribir *en memoria*, obteniendo lo escrito en un String
- ▶ **OutputStreamWriter**: flujo que permite transformar un OutputStream en un Writer.
- ▶ **PrintWriter**: flujo que permite escribir tipos básicos Java.

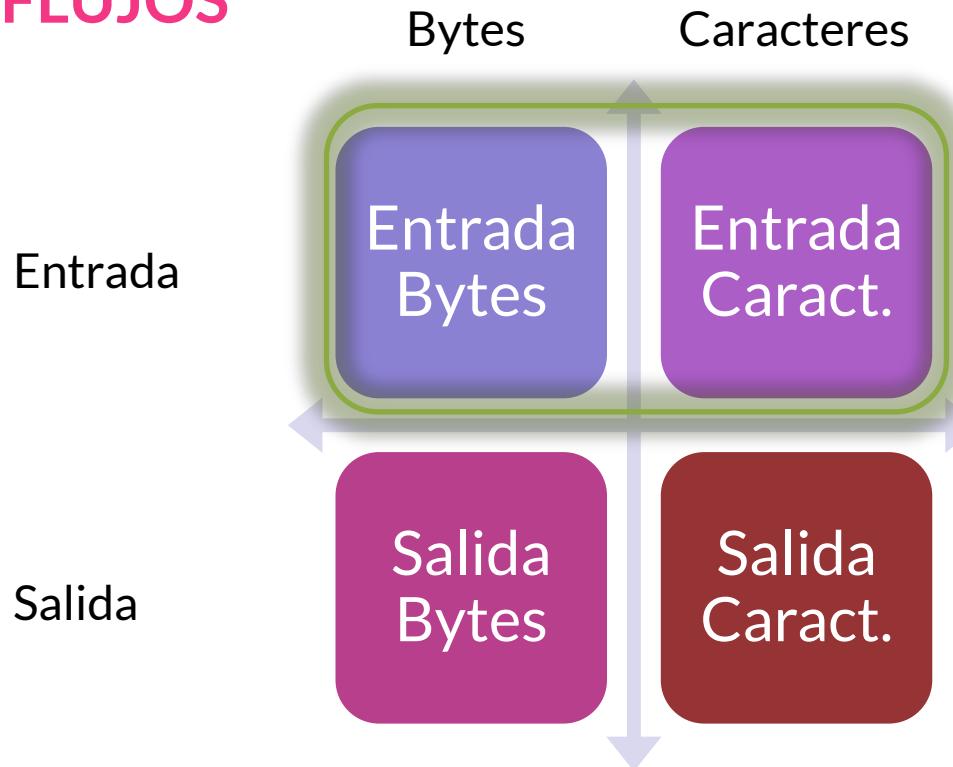
FLUJOS DE ENTRADA



FLUJOS DE ENTRADA



FLUJOS



FLUJOS DE ENTRADA

- ▶ Patrón básico de uso de flujos de entrada:

Abrir el flujo

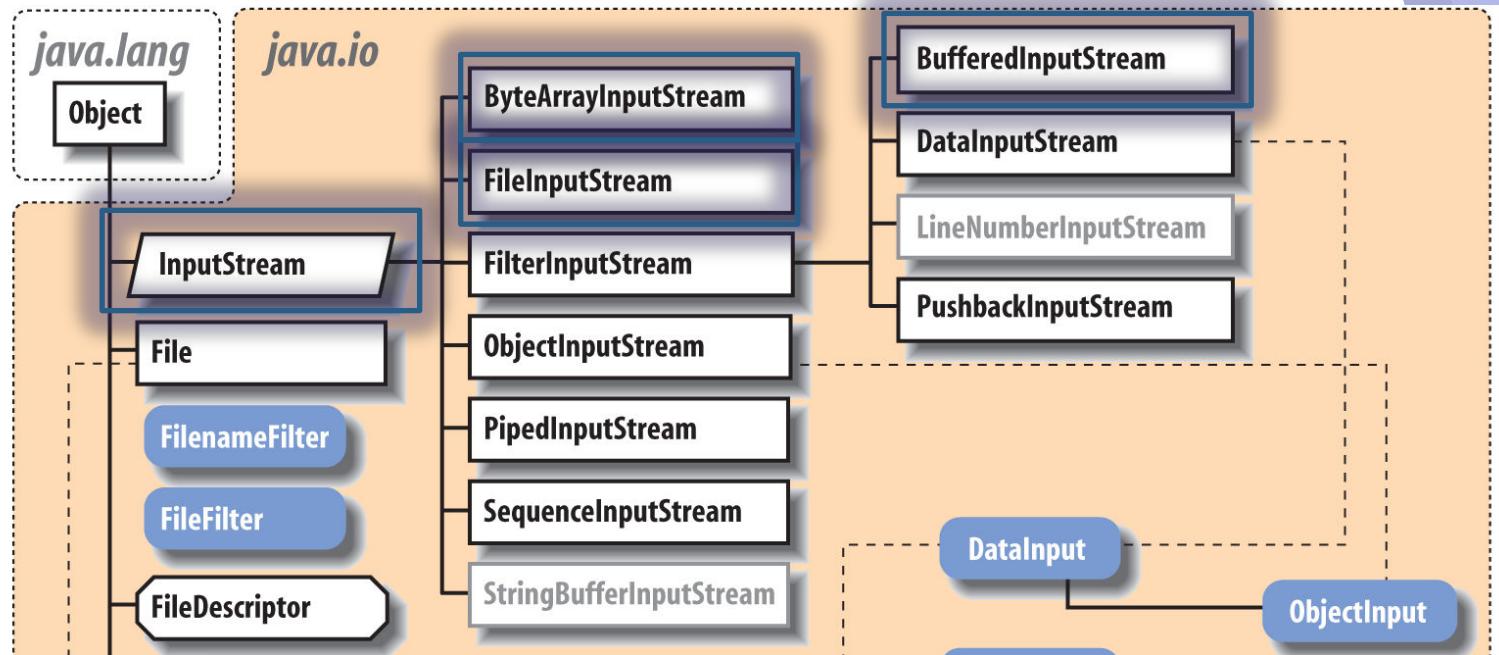
Mientras hay datos que leer

 Leer datos del flujo

 Procesarlos

Cerrar el flujo

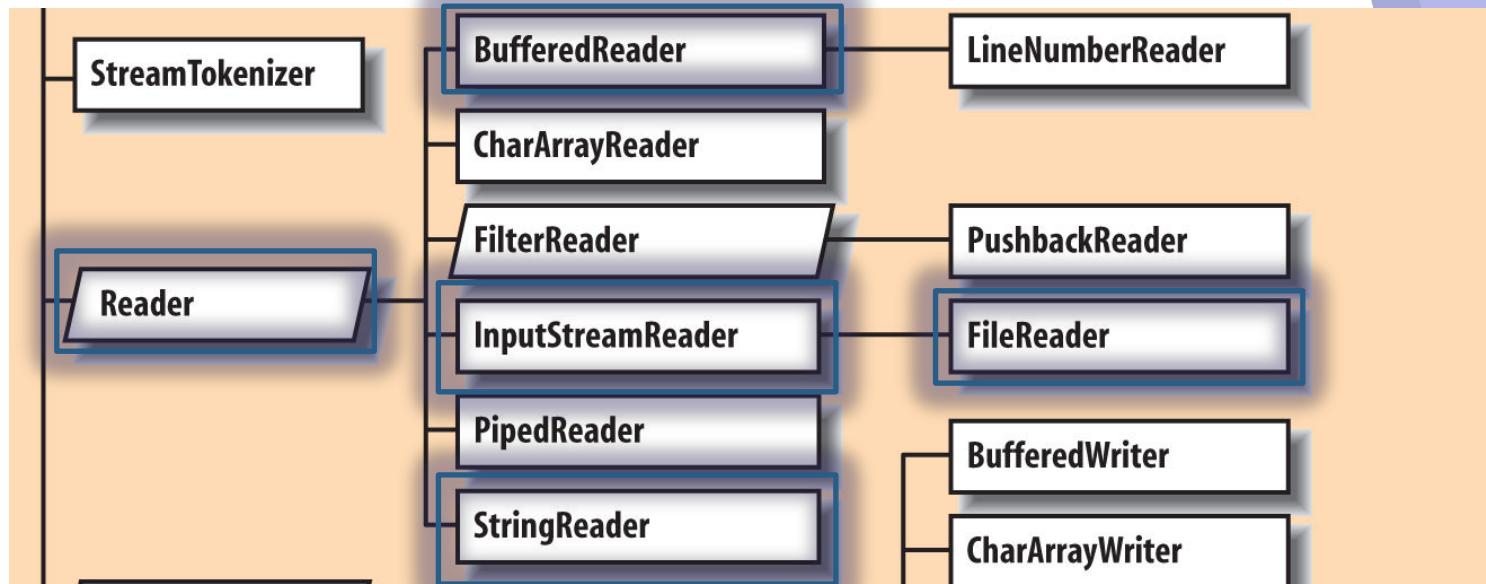
FLUJOS DE ENTRADA DE BYTES



FLUJOS DE ENTRADA DE BYTES

- ▶ ***InputStream***: clase abstracta, padre de la mayoría de los flujos de bytes.
- ▶ ***FileInputStream***: flujo que permite leer de un fichero, byte a byte.
- ▶ ***BufferedInputStream***: flujo que permite leer grupos (buffers) de bytes.
- ▶ ***ByteArrayInputStream***: flujo que permite leer de memoria (de un array de bytes).

FLUJOS DE ENTRADA DE CARACTERES



FLUJOS DE ENTRADA DE CARACTERES

- ▶ **Reader:** clase abstracta, padre de la mayoría de los flujos de caracteres.
- ▶ **FileReader:** flujo que permite leer de un fichero, carácter a carácter.
- ▶ **BufferedReader:** flujo que permite leer líneas de texto.
- ▶ **StringReader:** flujo que permite leer desde la memoria.
- ▶ **InputStreamReader:** flujo que permite transformar un InputStream en un Reader.



TRABAJAR CON LA CLASE FILE

CLASE FILE

- ▶ Fundamental hasta Java SE 6
- ▶ A partir de Java NIO.2 ha pasado a segundo plano por las ventajas de esta.
- ▶ Permite manejar ficheros y directorios.

```
File f = new File("./", "nuevo.txt");
f.createNewFile();
```

```
File temp = File.createTempFile("temporal", ".tmp");
System.out.println(temp.getAbsolutePath());
```

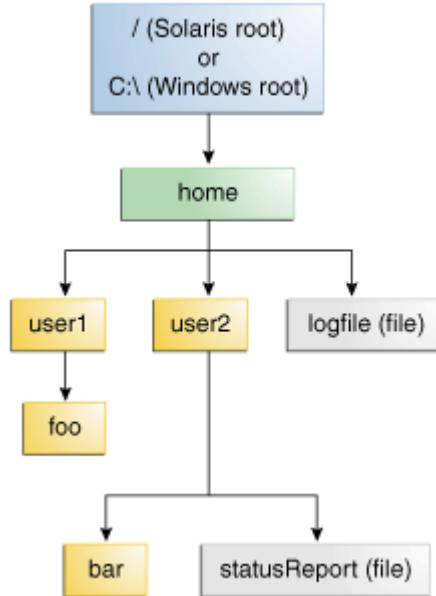
CLASE FILE

Nombre	Uso
isDirectory	Devuelve true si el File es un directorio
isFile	Devuelve true si el File es un fichero
createNewFile	Crea un nuevo fichero, si aun no existe.
createTempFile	Crea un nuevo fichero temporal
delete	Elimina el fichero o directorio
getName	Devuelve el nombre del fichero o directorio
getAbsolutePath	Devuelve la ruta absoluta del File
getCanonicalPath	Devuelve la ruta canónica del File
list, listFiles	Devuelve el contenido de un directorio



TRABAJANDO CON PATH

¿QUÉ ES UNA RUTA (PATH)?



Es la forma de acceder
(o identificar) un
fichero dentro del
sistema de ficheros

Unix/Solaris
/home/sally/statusReport

Windows
C:\home\sally\statusReport

ABSOLUTA O RELATIVA

- ▶ **Ruta absoluta:** aquella que contiene el elemento raiz y la lista de directorios completa para localizar el fichero
`/home/sally/statusReport`
- ▶ **Ruta relativa:** necesita ser combinada con otra para acceder al fichero (por ejemplo, nuestra *ubicación actual*).
`joe/foo`

INTERFAZ PATH

- ▶ Introducida en Java SE 7.
- ▶ Representa una ruta en el sistema de ficheros.
- ▶ Contiene el nombre de fichero y la lista de directorios usada para construir la ruta.
- ▶ Permite manejar diferentes sistemas de ficheros (Windows, Linux, Mac, ...)

OPERACIONES CON PATH

- ▶ Crear un Path
- ▶ Obtener información de un Path
- ▶ Eliminar redundancias
- ▶ Unir dos Paths
- ▶ Comparar dos paths
- ▶ ...

Paths

La creación de un *Path* hará casi siempre uso de los diferentes métodos estáticos de *Paths*



MÉTODOS DE LA CLASE FILES

FILES

- ▶ Tiene decenas de métodos estáticos para hacer múltiples operaciones con ficheros y directorios.

COMPROBACIONES

- ▶ Existencia (exists)
- ▶ Acceso (isReadable, isWritable, isExecutable)
- ▶ Son el mismo fichero (isSameFile)

COPIAR, BORRAR Y MOVER

- ▶ Borrar (delete, deleteIfExists)
- ▶ Copiar (copy)
- ▶ Mover (move)

CREAR, ESCRIBIR Y LEER

Para crear ficheros

- ▶ Ficheros regulares (createFile)
- ▶ Temporales (createTempFile)

Buffered

- ▶ Leer (newBufferedReader)
- ▶ Escribir (newBufferedWriter)

CREAR, ESCRIBIR Y LEER

Unbuffered

- ▶ Leer (newInputStream)
- ▶ Escribir (newOutputStream)

TRABAJO CON DIRECTORIOS

Listar

- ▶ Raíz del sistema
(FileSystem.getRootDirectories)
- ▶ Contenido de directorios
(newDirectoryStream)

Crear

- ▶ Crear un directorio (createDirectory,
createDirectories)
- ▶ Temporal (createTempDirectory)

USO DEL API STREAM CON JAVA NIO.2



NIO.2 + API STREAM

- ▶ La clase Files provee de algunos métodos *potentes* que devuelven Stream<T>.
- ▶ Podemos usarlos con lo que ya sabemos del API Stream para filtrar, mapear, reducir...

Files.list

- ▶ Devuelve todas las rutas de un directorio dado.
- ▶ De esta forma, podemos filtrar, mapear, reducir, ...

```
try (Stream<Path> stream =  
    Files.list(Paths.get(System.getProperty("user.home"), "ejemplo"))) {  
    stream  
        .map(String::valueOf)  
        .filter(path -> !path.startsWith("."))  
        .sorted()  
        .forEach(System.out::println);  
}
```

Files.find

- ▶ Devuelve todas las rutas a partir de un directorio dado que cumplen una condición. Se le puede indicar una profundidad máxima.

```
Path start = Paths.get(System.getProperty("user.home"), "ejemplo");
int maxDepth = 5;
try (Stream<Path> stream = Files.find(start, maxDepth, (path, attr) ->
    String.valueOf(path).endsWith(".txt"))) {
    stream
        .sorted()
        .map(String::valueOf)
        .forEach(System.out::println);
}
```

Files.walk

- ▶ Devuelve todas las rutas a partir de un directorio dado. Se le puede indicar una profundidad máxima.

```
Path start = Paths.get(System.getProperty("user.home"), "ejemplo");
int maxDepth = 5;
try (Stream<Path> stream = Files.walk(start, maxDepth)) {
    TreeMap<String, Long> groupByExtension =
        stream.filter(Files::isRegularFile)
        .sorted()
        .collect(Collectors.groupingBy(C_Walk::getExtension, TreeMap::new,
                                      Collectors.counting()));

    groupByExtension.forEach((k, v)
        -> System.out.printf("%s -> %d ficheros%n", k, v));
}
```

Files.lines

- ▶ Devuelve las líneas de un fichero de texto en un Stream.

```
try (Stream<String> stream = Files.lines(p, Charset.forName("Cp1252"))) {  
    return Optional.of(stream  
        .map(s -> s.split(";"))  
        .collect(Collectors.toList()));  
}
```

PROGRAMACIÓN CONCURRENTE



MULTITAREA

- ▶ Sistema que permite realizar dos o más tareas simultáneas (aparentemente).
- ▶ Un procesador *mononúcleo* solo es capaz de ejecutar una tarea en un momento determinado.
- ▶ ¿Cómo hace para ejecutar varias tareas a la vez?

MULTIPROCESAMIENTO

Diferencias entre programa y proceso

- ▶ **Programa:** conjunto de instrucciones y datos. Es un ente estático. Un texto que indica qué hacer con unas variables
- ▶ **Proceso:** es un programa en ejecución, un ente dinámico. Lleva asociado un estado de ejecución (registros, contador de programa, ...:bloque de control de proceso)

MULTIPROCESAMIENTO

- ▶ Dos procesos son concurrentes (hay multiprocesamiento) cuando haya un solapamiento de la ejecución de sus instrucciones
- ▶ *La 1^a instrucción de uno de ellos se ejecuta después de la 1^a instrucción del otro y antes de la última.*
- ▶ El soporte para multiprocesamiento nos lo da el sistema operativo.

HILO

- ▶ En programación un hilo es un componente de un proceso.
- ▶ Tienen su propia pila, sus propios valores de registros y valor del contador de programa.
- ▶ Un proceso tiene al menos un hilo, pero puede tener más.
- ▶ Permiten que un proceso *haga más de una tarea a la vez.*

MULTIPROCESO vs. MULTIHILO

Los procesos son entes pesados, situados en espacios de memoria diferentes

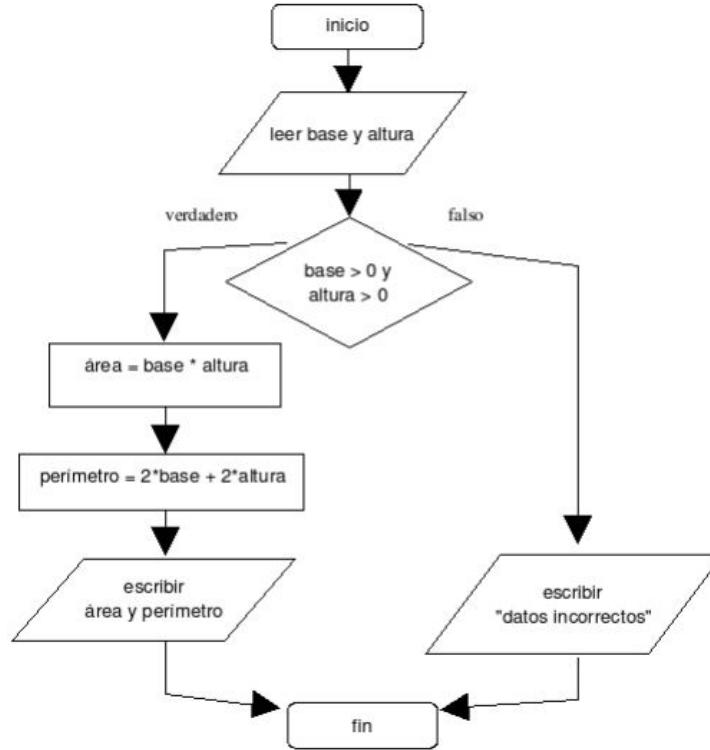
- ▶ Dificultad de comunicación
- ▶ Lentitud en el cambio de contexto

Los hilos son entes ligeros, situados en el mismo espacio de memoria.

- ▶ Comparten espacio de memoria
- ▶ Tienen su propia pila, variables y CP
- ▶ Cambios de contexto más rápidos.

PROGRAMAS DE FLUJO ÚNICO

- ▶ Usan un único control de flujo para controlar la ejecución.
- ▶ Se ejecutan en un solo hilo.
- ▶ Java crea y destruye el mismo.



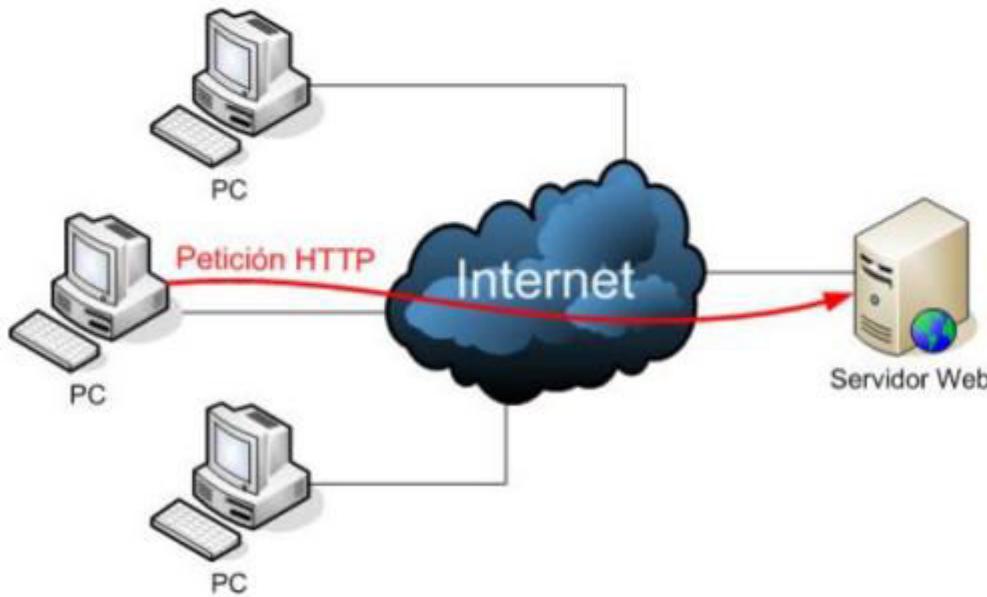
PROGRAMAS DE FLUJO MÚLTIPLE

Durante su ejecución, la aplicación necesita realizar varias tareas a la vez.



PROGRAMAS DE FLUJO MÚLTIPLE

Durante su ejecución, la aplicación necesita realizar varias tareas a la vez.



SOPORTE DE JAVA SE 8

- ▶ Java permite programación con multiprocesamiento y multihilo.
- ▶ Nos centraremos en la segunda, dadas las ventajas que nos ofrece, y por ser la más utilizada.

HILOS Y SU CICLO DE VIDA



CLASE THREAD

- ▶ Cada hilo de ejecución de nuestras aplicaciones se asocia a una instancia de Thread.

Sus métodos básicos son:

- ▶ ***public void run()***: contiene el código que queremos ejecutar en el hilo. NO se debe invocar nunca directamente.
- ▶ ***public void start()***: lanza la ejecución del hilo.

CLASE THREAD

Código que se ejecutará en otro hilo

```
public class PrimoThread extends Thread {  
    //propiedades y constructor  
    public void run() {  
        long n = minimo;  
        while(!testPrimalidad(n)) {  
            System.out.printf("%d no es primo %n", n);  
            ++n;  
        }  
        System.out.printf("El número primo es %d %n", n);  
    }  
}
```

```
public static boolean testPrimalidad(long n) {  
    //cuerpo del método  
}  
}
```

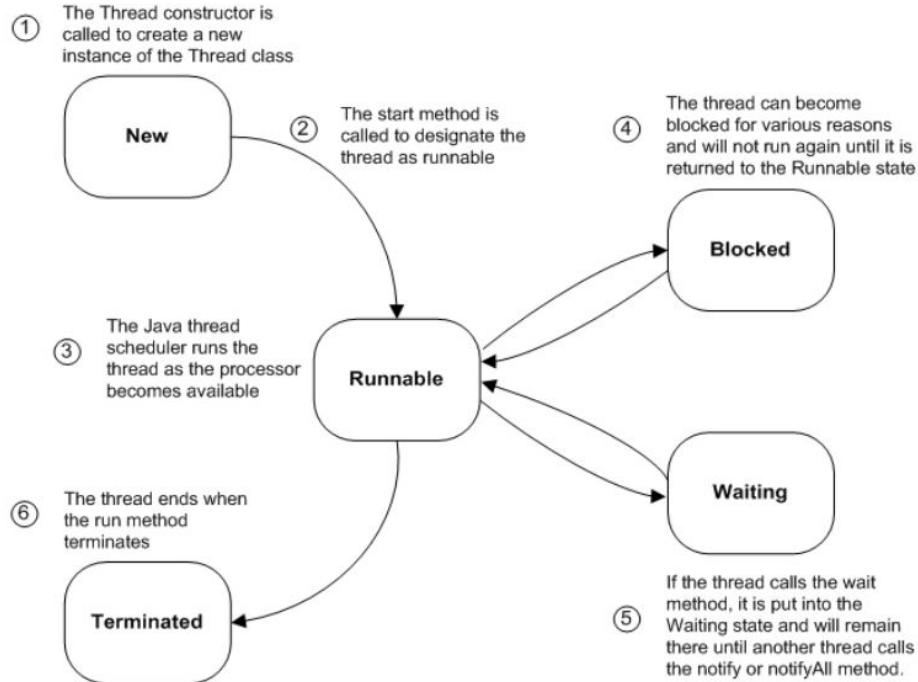
EJECUCIÓN DE UN THREAD

```
public class Ejemplo02Primo {  
  
    public static void main(String[] args) {  
  
        PrimoThread pt01 = new PrimoThread(1234567);  
        pt01.start();  
        PrimoThread pt02 = new PrimoThread(23456789);  
        pt02.start();  
        PrimoThread pt03 = new PrimoThread(34567891);  
        pt03.start();  
  
    }  
  
}
```

El método
run() no se
invoca
directamente

La ejecución
de los 3 hilos
se solapará

CICLO DE VIDA DE UN THREAD



CICLO DE VIDA DE UN THREAD

1. Se llama al constructor de Thread para crear el nuevo hilo.
2. Se llama al método **start** para designarlo como ejecutable
3. El planificador lo ejecuta en cuanto el procesador está disponible.

CICLO DE VIDA DE UN THREAD

4. El hilo puede pasar a **bloqueado** por diferentes razones, y no vuelve hasta que pasa de nuevo a **ejecutable**.
5. Si se utiliza el método **wait**, se pone en estado de espera, y permanece ahí hasta que se ejecuta **notify** o **notifyAll**.
6. El hilo termina cuando finaliza la ejecución de su método **run**.

PAUSAR LA EJECUCIÓN DE UN THREAD

- ▶ Usando el método `Thread.sleep(long millis)`.
- ▶ Es un método estático
- ▶ Pausa el hilo que actualmente está ejecutándose.

RUNNABLE, CALLABLE Y EXECUTORSERVICE



1.

RUNNABLE

RUNNABLE

- ▶ Si nuestra clase ya hereda de una, no puede heredar de *Thread*.
- ▶ Runnable es un interfaz que nos permite crear tareas para ser ejecutadas en hilos secundarios.
- ▶ *Thread* tiene un constructor que permite pasar como argumento un *Runnable*.

```
public interface Runnable {  
    public void run();  
}
```

RUNNABLE

```
public class HelloRunnable implements Runnable {  
    public void run() {  
        System.out.println("Hello from a thread!");  
    }  
  
    public static void main(String args[]) {  
        (new Thread(new HelloRunnable())).start();  
    }  
}
```

2.

CALLABLE

CALLABLE<V>

- ▶ Runnable (o Thread) no permiten devolver valores.
- ▶ Complejos mecanismos de sincronización para hacerlo.
- ▶ Callable es, básicamente, un Runnable que puede devolver un valor.

```
public interface Callable<V> {  
    public V call();  
}
```

CALLABLE<V>

```
public class PrimoCallable implements Callable<Long> {  
  
    private long minimo;  
  
    public PrimoCallable(long minimo) {  
        this.minimo = minimo;  
    }  
  
    @Override  
    public Long call() throws Exception {  
        long n = minimo;  
        System.out.println("Comenzamos a buscar un número primo");  
        while(!testPrimalidad(n)) {  
            System.out.printf("%d no es primo %n", n);  
            ++n;  
        }  
        return n;  
    }  
}
```

FUTURE<V>

- ▶ Interfaz que representa el resultado de una computación asíncrona.
- ▶ Nos permite algunas operaciones: comprobar el resultado, saber si la computación ha terminado, esperar a que termine, ...
- ▶ Método `get` para obtener el valor de la ejecución de un **Callable<V>**.
- ▶ Nos invita a usar **Executor**

3. EXECUTORs

MANEJO DE HILOS A ALTO NIVEL

- ▶ Hasta ahora, el programador definía y lanzaba los hilos de ejecución según su necesidad.
- ▶ Válido para aplicaciones pequeñas.
- ▶ Para grandes aplicaciones, hay que separar la administración de los hilos del resto de la aplicación.
- ▶ Esto lo podemos realizar mediante ejecutores (Executors).

EXECUTORs

3 interfaces

- ▶ *Executor*: soporta el lanzamiento de nuevas tareas, bajo demanda.
- ▶ ***ExecutorService***: añade a la anterior características que permiten administrar el ciclo de vida.
- ▶ ***ScheduledExecutorService***: añade a la anterior la posibilidad de ejecutar tareas periódicas.

EXECUTORSERVICE

- ▶ `submit(...)` acepta *Runnable* o *Callable*.
- ▶ Métodos para la finalización del propio ejecutor.
- ▶ Creación a partir de un pool de hilos que haga uso de *worker threads*: hilos que son reutilizables, minimizando la sobrecarga de la creación de hilos nuevos.
- ▶ Podemos finalizar el ejecutor con el método *shutdown*.

POOLS DE HILOS

Single

- ▶ Con un solo hilo de ejecución disponible.
- ▶ Si le pedimos (*submit*) más de una tarea a la vez, las pone en cola.

Fixed

- ▶ Indicamos, en el momento de su creación, el número de hilos.
- ▶ Si dispone de n hilos, y enviamos $n+1$ tareas, las pone en cola.

POOLs DE HILOS

Cached

- ▶ Crea hilos conforme enviamos tareas
- ▶ Reutiliza los hilos cuyas tareas han finalizado, para ejecutar tareas nuevas.

CREACIÓN DE POOLS DE HILOS

La clase Executors tiene métodos estáticos para construir cada tipo. Entre ellos

- ▶ `newSingleThreadExecutor()`: crea un ejecutor de tipo *single*.
- ▶ `newFixedThreadPool(int n)`: crea un ejecutor de tipo *fixed* con *n* hilos disponibles.
- ▶ `newCachedThreadPool()`: crea un ejecutor de tipo *cached*.

CÓDIGO SINCRONIZADO Y USO DE VALORES ATÓMICOS



INTERFERENCIA ENTRE HILOS

- ▶ ¿Qué sucede si dos hilos acceden a la misma variable a la vez, alguno de ellos en modo escritura?
- ▶ Una sentencia java es, a bajo nivel, varias instrucciones del procesador.
- ▶ Condición de carrera: *no podemos asumir el orden en el que el procesador va a ejecutar las instrucciones de dos hilos diferentes.*

SYNCHRONIZED

- ▶ Métodos o bloques.
- ▶ Indican que ese bloque de código (o método) puede ser ejecutado solo por un hilo a la vez. El otro debe esperar.

```
public synchronized void incrementar() {  
    c++;  
}  
  
synchronized(this) {  
    count++;  
}
```

CERROJOS INTRÍNSECOS O MONITOR

- ▶ Entidad interna que permite la sincronización.
- ▶ Cada objeto tiene su propio cerrojo intrínseco.
- ▶ Un método sincronizado adquiere el cerrojo de un objeto, y lo libera cuando termina.
- ▶ Si un hilo tiene el cerrojo de un objeto compartido, otro no puede adquirirlo hasta que el primero lo libera.

VARIABLES ATÓMICAS

- ▶ En el paquete `java.util.concurrent.atomic`
- ▶ Tipos básicos que ya están sincronizados (*thread-safe*).
- ▶ Métodos `get`, `set`, `counterAndSet`, ...
- ▶ `AtomicBoolean`, `AtomicInteger`, `AtomicLong`,
`LongAdder`, `DoubleAdder`, ...

<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html>

CONEXIÓN CON UNA BASE DE DATOS



JDBC

- ▶ Java soporta la conexión con base de datos, en especial, de tipo relacional.
- ▶ JDBC = Java DataBase Connectivity
- ▶ Java SE 8 nos da la versión JDBC 4.2 (JSR 211), y ofrece una serie de interfaces.
- ▶ Conectividad con múltiples bases de datos de terceros a través de *drivers*.

TIPOS DE DRIVERS

Tipo 1	<i>Bridge</i> : sirve de puente con otro API, como ODBC
Tipo 2	<i>Native</i> : traduce Java al API nativo de la base de datos
Tipo 3	<i>Middleware</i> : utiliza un servidor intermedio y un protocolo estándar.
Tipo 4	<i>Pure</i> : Se conecta directamente a la base de datos a través de su protocolo de comunicaciones.

Lista de drivers disponibles para JDBC

<http://www.oracle.com/technetwork/java/index-136695.html>

Driver para Mysql (Connector/J 5.1)

<https://dev.mysql.com/downloads/connector/j/5.1.html>

SQL

- ▶ Para trabajar con JDBC es necesario que manejemos SQL (Structured Query Language)
- ▶ Lenguaje de consulta y manipulación de datos de cualquier base de datos relacional.

```
SELECT ...
FROM ...
WHERE ...
;
```

INTERFACES PRINCIPALES

- ▶ *Connection*: es el que permite mantener la conexión con la base de datos.
- ▶ *Statement, PreparedStatement*: nos permiten ejecutar consultas.
- ▶ *ResultSet*: juego de resultados de una consulta ejecutada.

CONEXIÓN CON LA BASE DE DATOS

2 opciones

- ▶ *DriverManager*: nos permite conectar con una base de datos a través de una url jdbc.
- ▶ *DataSource*: interfaz más avanzado, permite ser transparente a nuestra aplicación. Más complejo que *DriverManager*.

NUESTRA ELECCIÓN

A lo largo de estos capítulos utilizaremos **DriverManager**. Aunque no es obligatorio, el uso de *DataSource* está orientado a proyectos Java EE.

DriverManager será suficientemente potente para *pequeños proyectos*. Para otros más grandes, sería *recomendable* usar un sistema de persistencia como *JPA*.

URL JDBC

- ▶ Una cadena de texto con los datos de conexión a la base de datos concreta
- ▶ Depende del driver/base de datos

MySQL	Driver: com.mysql.jdbc.Driver URL: jdbc:mysql://hostname/database
Oracle	Driver: oracle.jdbc.driver.OracleDriver URL: jdbc:oracle:thin@hostname:port:database

PASOS PARA CONECTAR

1. Cargar el driver JDBC (< 4.0)
2. Establecer datos de conexión
3. Conectar obteniendo un objeto *Connection*.
4. Crear un objeto *Statement* y ejecutar consultas SQL
5. Los resultados se almacenan en un objeto *ResultSet*, donde se pueden consultar.
6. Cerrar los objetos (*ResultSet*, *Statement* y *Connection*).

**LANZAR
CONSULTAS Y
PROCESAR
RESULTADOS**



STATEMENT

- ▶ Nos provee de métodos para ejecutar consultas en la base de datos.
- ▶ Recibe las consultas como un *String*.
- ▶ Genera, tras la ejecución de una consulta, un objeto de tipo *ResultSet*.

execute	Para obtener más de un ResultSet
executeQuery	Devuelve un solo ResultSet
executeUpdate	Devuelve un entero que representa el número de filas afectadas. Se usa con consultas INSERT, UPDATE o DELETE

RESULTSET

- ▶ Recoge los resultados que devuelve una consulta.
- ▶ Tiene una estructura de *cursor*.
- ▶ Podemos *navegar* fila a fila, extrayendo los resultados con los métodos `getXXX`.
- ▶ Tiene métodos para todos los tipos de datos básicos que podemos utilizar en nuestras tablas.

RESULTSET

- ▶ El métodos *next()* devuelve true en tanto en cuanto existen más resultados.

```
while (rs.next()) {  
    String coffeeName = rs.getString(1);  
    int supplierID = rs.getInt(2);  
    float price = rs.getFloat(3);  
    int sales = rs.getInt(4);  
    int total = rs.getInt(5);  
    //...  
}
```

RESULTSET

Tiene otros métodos para navegar por el cursor:

- ▶ *previous*
- ▶ *first*
- ▶ *last*
- ▶ *beforeFirst*
- ▶ *afterLast*
- ▶ *relative(int)*
- ▶ *absolute(int)*

RESULTSET en MODO ESCRITURA

- ▶ Solo algunos drivers soportan los *ResultSet* en modo escritura.
- ▶ Es preferible trabajar con otros esquemas, como el uso de *PreparedStatement* y el patrón de diseño DAO (Data Access Object).

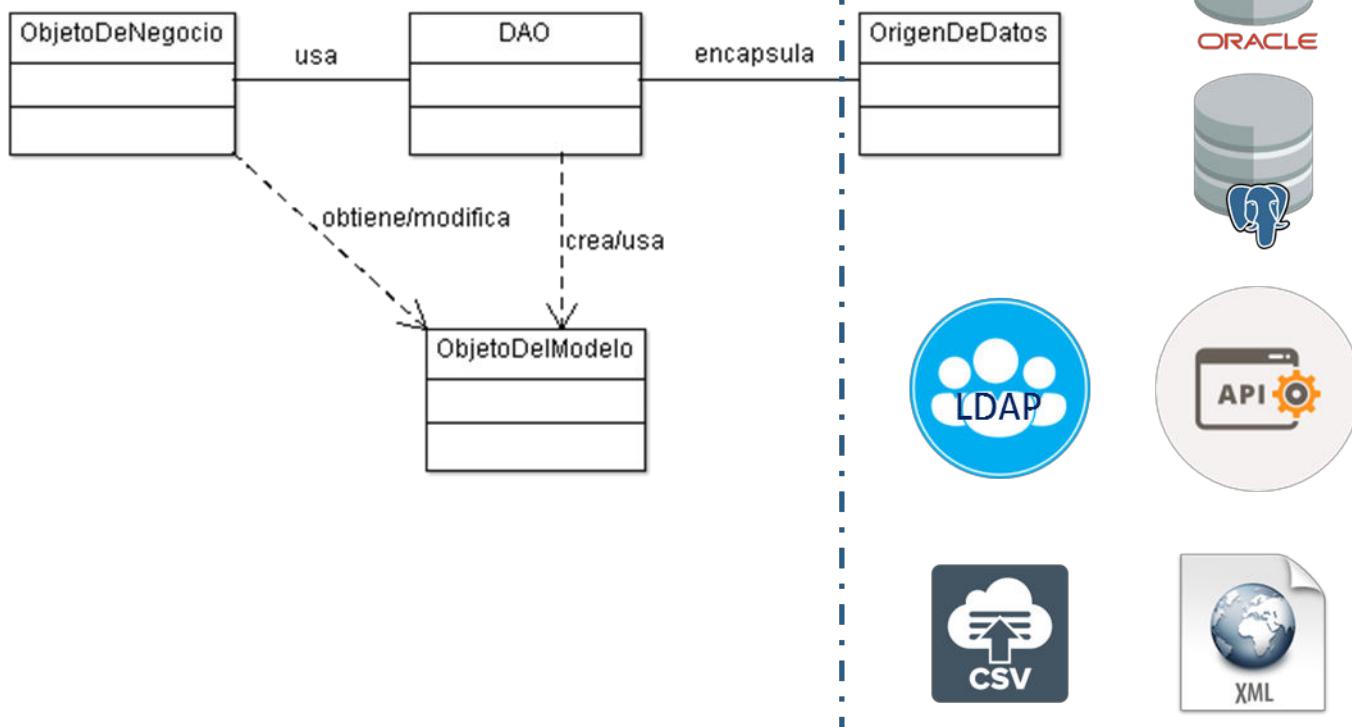
PREPAREDSTATEMENT

- ▶ Una extensión de Statement.
- ▶ Nos permite prever problemas de *inyección de SQL*.
- ▶ En lugar de contactaran los parámetros en una consulta SQL, indicamos los «huecos» y posteriormente le asignamos los valores.
- ▶ JDBC se encarga de «precompilar» la consulta antes de enviarla, evitando código malicioso.

PATRÓN DAO

- ▶ Patrón de diseño de software.
- ▶ Uso de clases *modelo*.
- ▶ Un solo objeto se encarga de realizar las operaciones con la base de datos.
- ▶ El resto del sistema trabajar con ese objeto, que nos aísla del *sgbd* concreto.

PATRÓN DAO





USO DE ROWSET

ROWSET

- ▶ Se trata de un objeto que permite manejar información *tabular* de forma más flexible y fácil que un *ResultSet*.
- ▶ Existen 5 tipos distintos: *JdbcRowSet*, *CachedRowSet*, *JoinRowSet*, *FilteredRowSet* y *WebRowSet*.

CAPACIDADES DE UN ROWSET

¿En qué se diferencian de un *ResultSet*?

- ▶ Puede funcionar como un JavaBean, así que puede hacer uso de propiedades (properties) y del mecanismo de notificación de los JavaBeans (listeners).
- ▶ Posibilidad de actualizar valores y hacer scroll, independientemente del DBMS utilizado.

TIPOS DE ROWSET

Según si están o no conectados:

- ▶ **Conectados:** el RowSet siempre tiene la conexión abierta con la base de datos.
JdbcRowSet
- ▶ **Desconectados:** la conexión no siempre está abierta, así que son más ligeros y serializables. *CachedRowSet, JoinRowSet, FilteredRowSet, WebRowSet.*

CREACIÓN DE UN ROWSET

Desde Java 1.7 la mejor forma es a través de la correspondiente factoría.

```
RowSetFactory myRowSetFactory = null;  
JdbcRowSet rowSet = null;  
  
myRowSetFactory = RowSetProvider.newFactory();  
rowSet = myRowSetFactory.createJdbcRowSet();
```

La factoría nos permite crear cualquier tipo de RowSet.

JDBCROWSET

- ▶ Mantiene siempre la conexión abierta a la base de datos (rápido e ineficiente).
- ▶ Nos permite recorrer los resultados, actualizarlos, insertar nuevos, borrar, etc...

CACHEDROWSET

- ▶ Solo abre la conexión en momento puntuales.
- ▶ Tenemos que indicar que columnas son la clave (primaria).
- ▶ Los cambios tienen que ser aceptados (`acceptChanges`). Requiere `autocommit = false`;
- ▶ Nos permite realizar las mismas operaciones que con un `JdbcRowSet`.