IES DR LLUÍS SIMARRO

ENSENYAMENT PÚBLIC

XÀTIVA

CFGS Desarrollo de aplicaciones web

Módulo profesional: Programación

GENERALITAT VALENCIANA

Conselleria d'Educació, Investigació, Cultura i Esport

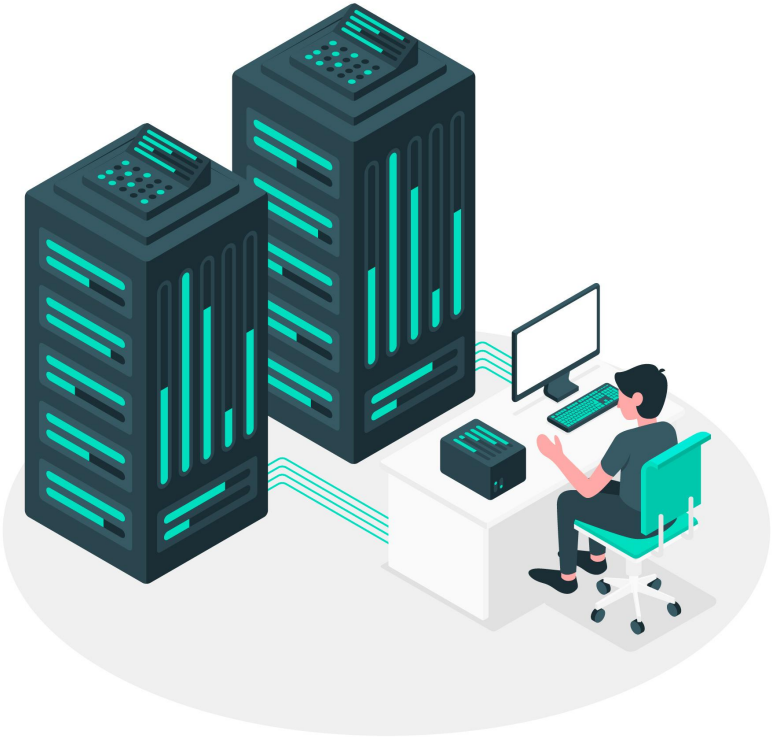
GOBIERNO DE ESPAÑA

MINISTERIO DE EDUCACIÓN Y FORMACIÓN PROFESIONAL

Unió Europea

Fons Social Europeu

L'FSE inverteix en el teu futur



Material elaborado por:

Anna Sanchis

Carlos Cacho

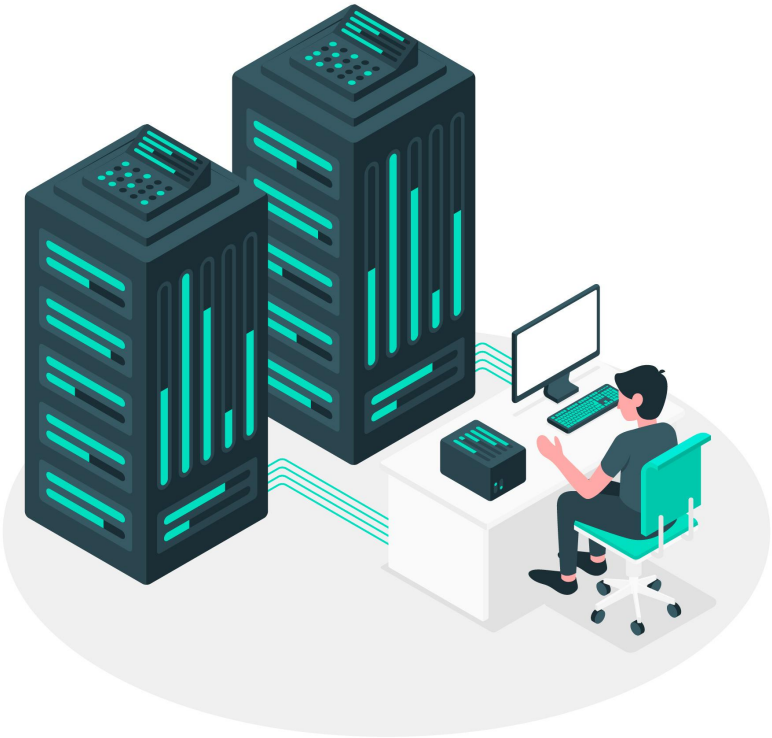
Raquel Torres

Lionel Tarazón

Fco. Javier Valero

Revisado y editado por:

Edu Torregrosa Llácer



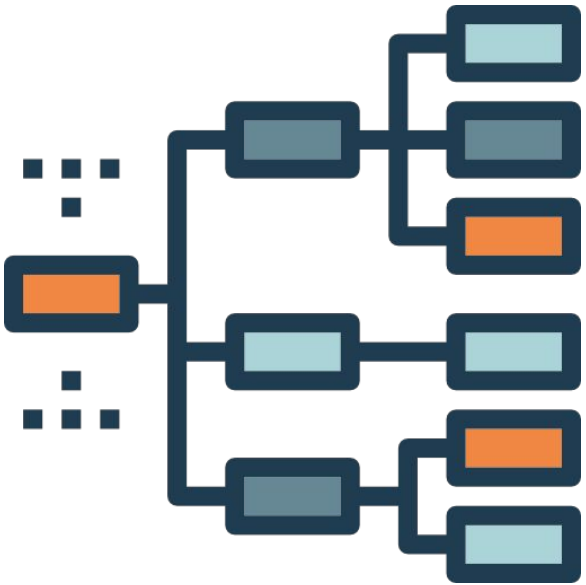
Datos profesor

Edu Torregrosa Llácer

eduardotorregrosa@ieslluissimarro.org

Web del módulo: https://portal.edu.gva.es/aules/

Herencia y excepciones en JAVA



1. Introducción.
2. Herencia
3. Interfaces
4. Excepciones

Introducción

- La Herencia es uno de los 4 pilares de la programación orientada a objetos junto con la Abstracción, Encapsulación y Polimorfismo.
  - → **Abstracción**: Capacidad de expresar las características específicas de un objeto, aquellas que lo distinguen de los demás tipos de objetos.
  - → **Encapsulación**: Capacidad del objeto de responder a peticiones a través de sus métodos sin la necesidad de exponer los medios utilizados para llegar a brindar estos resultados.
  - → **Herencia**: es el mecanismo que permite la definición de una clase a partir de la definición de otra ya existente. La herencia permite compartir automáticamente métodos y datos entre clases.
  - → **Polimorfismo**: Polimorfismo significa que la misma operación puede comportarse diferentemente sobre distintas clases.

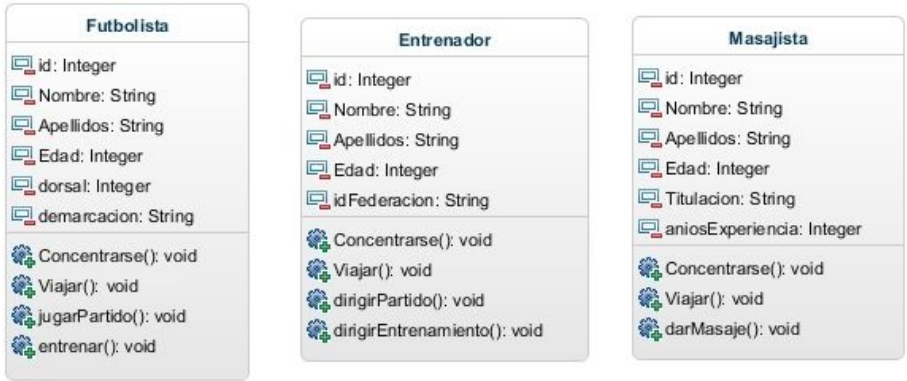
Introducción

- Abstracción:

  → ¿Cómo se consigue?

  Al incluir en una misma clase tanto los atributos como los métodos.

Vamos a definir tres clases que van a representaran a objetos Futbolista, Entrenador y Masajista. De cada unos de ellos vamos a necesitar algunos datos que reflejaremos en los **atributos** y una serie de acciones que reflejaremos en sus **métodos**.



Introducción

- **Encapsulamiento**:
  - Se consigue con métodos y modificadores **public, private, protected**

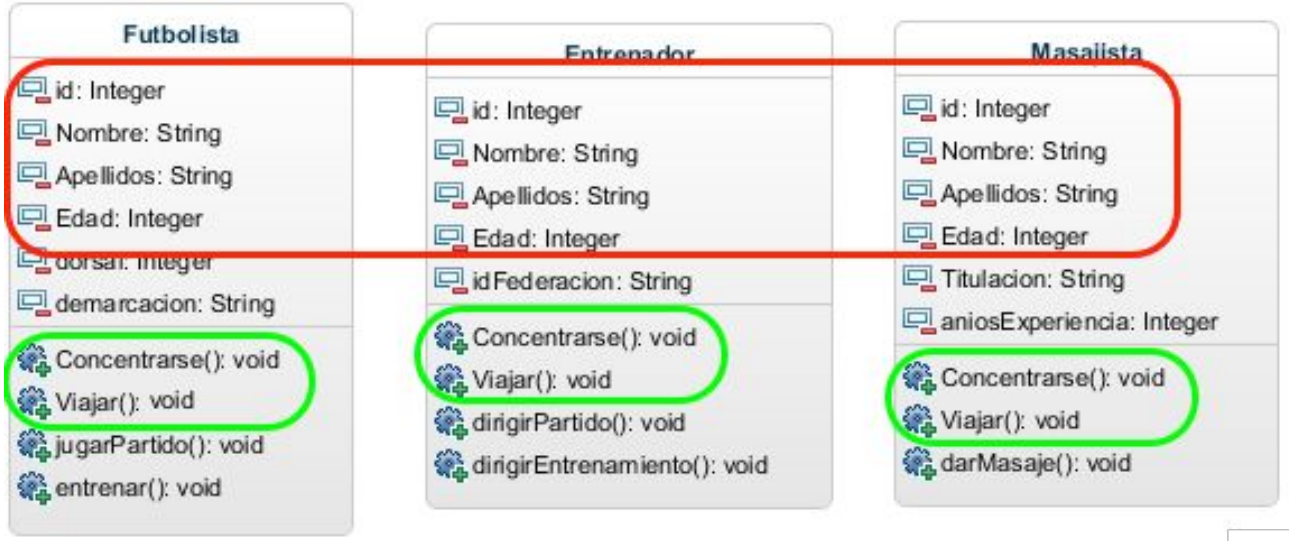
```java
public class Futbolista
{

        private int id;
        private String Nombre;
        private String Apellidos;
        private int Edad;
        private int dorsal;
        private String demarcacion;

        // constructor, getter y setter

        public void Concentrarse() {
                ...
        }

        public void Viajar() {
                ...
        }

        public void jugarPartido() {
                ...
```

```java
public class Entrenador
{

        private int id;
        private String Nombre;
        private String Apellidos;
        private int Edad;
        private String idFederacion;

        // constructor, getter y sette
r

        public void Concentrarse() {
                ...
        }

        public void Viajar() {
                ...
        }

        public void dirigirPartido() {
                ...
```

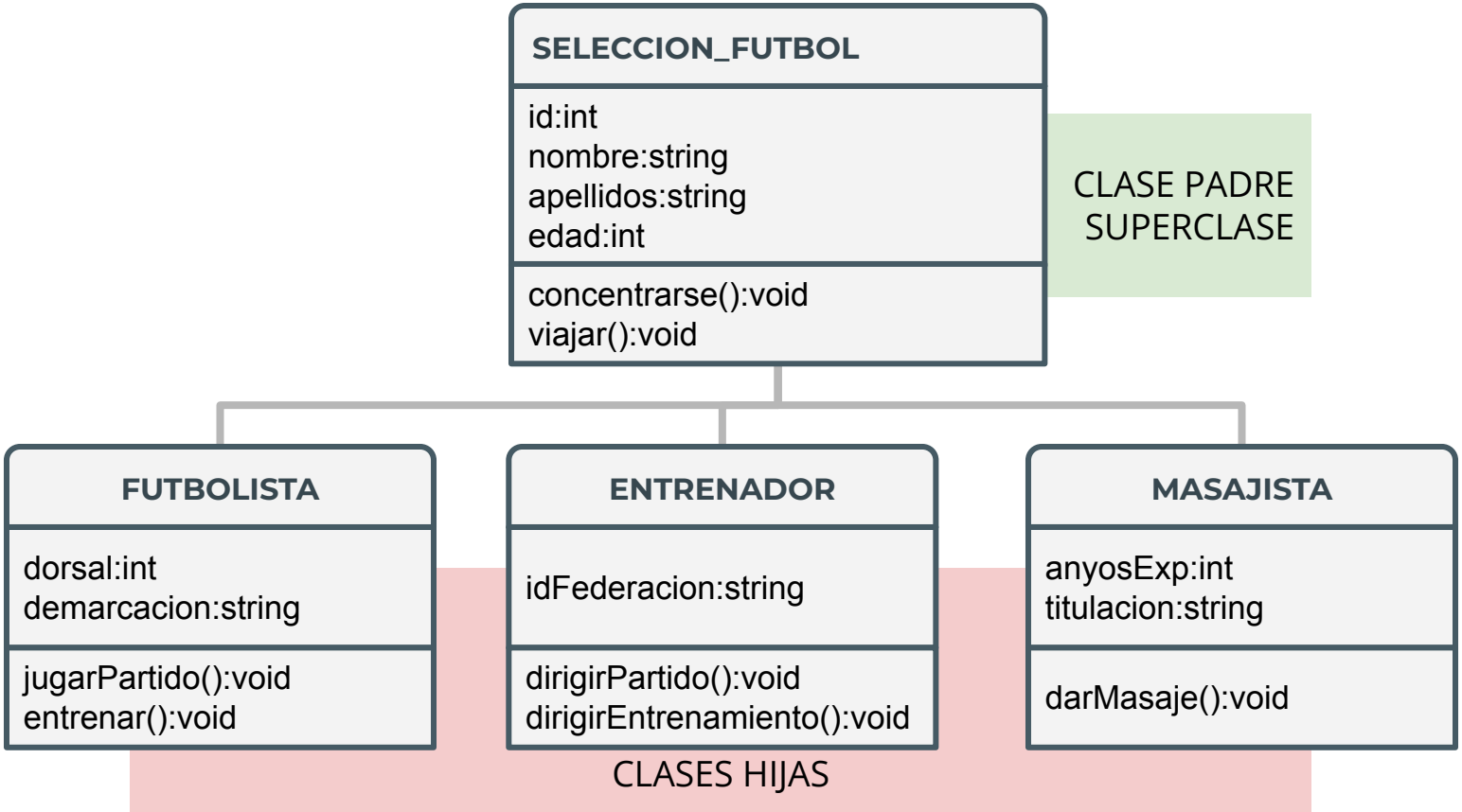
```java
public class Masajista
{

        private int id;
        private String Nombre;
        private String Apellidos;
        private int Edad;
        private String Titulacion;
        private int aniosExperiencia;

        // constructor, getter y setter

        public void Concentrarse() {
                ...
        }

        public void Viajar() {
                ...
        }

        public void darMasaje() {
                ...
```

Herencia

- **Herencia**:
  - → Vemos que en las tres clases tenemos **atributos** y **métodos que son iguales** ya que los tres tienen los atributos id, Nombre, Apellidos y Edad; y los tres tienen los métodos de Viajar y Concentrarse:

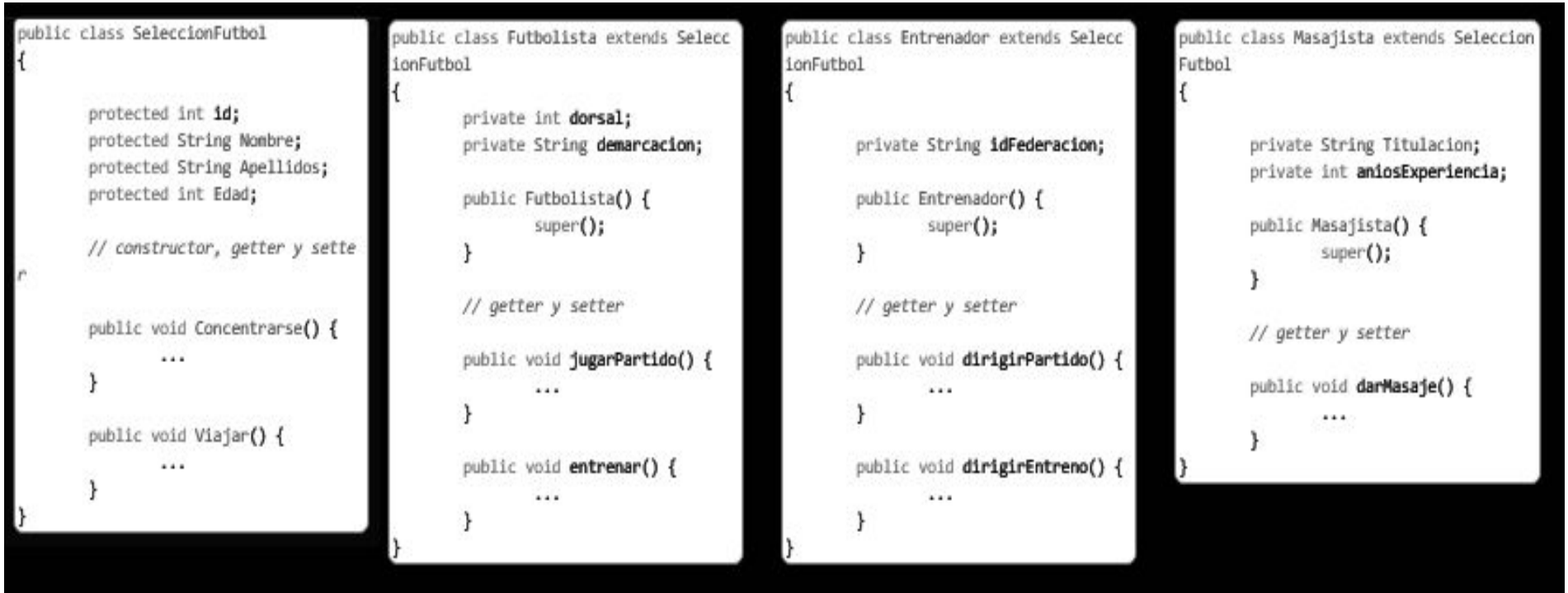


Herencia


SELECCION_FUTBOL
id:int
nombre:string
apellidos:string
edad:int
concentrarse():void
viajar():void
CLASE PADRE
SUPERCLASE
FUTBOLISTA
dorsal:int
demarcacion:string
jugarPartido():void
entrenar():void
ENTRENADOR
idFederacion:string
dirigirPartido():void
dirigirEntrenamiento():void
MASAJISTA
anyosExp:int
titulacion:string
darMasaje():void
CLASES HIJAS

Herencia en JAVA

- La herencia en Java se realiza utilizando la palabra reservada extends



```
public class SeleccionFutbol
{

        protected int id;
        protected String Nombre;
        protected String Apellidos;
        protected int Edad;

        // constructor, getter y sette
r

        public void Concentrarse() {
                ...
        }

        public void Viajar() {
                ...
        }
}
```

```
public class Futbolista extends Selecc
ionFutbol
{
        private int dorsal;
        private String demarcacion;

        public Futbolista() {
                super();
        }

        // getter y setter

        public void jugarPartido() {
                ...
        }

        public void entrenar() {
                ...
        }
}
```

```
public class Entrenador extends Selecc
ionFutbol
{

        private String idFederacion;

        public Entrenador() {
                super();
        }

        // getter y setter

        public void dirigirPartido() {
                ...
        }

        public void dirigirEntreno() {
                ...
        }
}
```

```
public class Masajista extends Seleccion
Futbol
{

        private String Titulacion;
        private int aniosExperiencia;

        public Masajista() {
                super();
        }

        // getter y setter

        public void darMasaje() {
                ...
        }
}
```

Herencia en JAVA

- **Cambiamos la visibilidad de los atributos.**
- Los **atributos** pueden ser de **tres tipos**, que definen su grado de comunicación y visibilidad con el entorno, estos son:
  - → **public** (+): el atributo será visible dentro y fuera de la clase.
  - → **private** (-): el atributo sólo será accesible desde dentro de la clase (sólo sus métodos pueden manipular los atributos)
  - → **protected** (#): el atributo no será accesible desde fuera de la clase, pero sí podrá ser manipulado por métodos de la clase y de sus subclases. (Acceso público para las clases derivadas y acceso privado (prohibido) para el resto de clases.
  - → **package**(): Se puede acceder al atributo desde cualquier clase del package donde se define la clase
- La herencia en Java no modificar la visibilidad de los atributos de la clase base. Es decir, un atributo public en la clase padre también lo será en la clase hija.

Herencia en JAVA

- **Genera código mucho más limpio, estructurado y con menos líneas de código, lo que lo hace más legible** .
- Si queremos añadir más clases a nuestra aplicación, en la clase padre (SeleccionFutbol) tenemos implementado parte de sus datos y de su comportamiento y solo habrá que implementar los atributos y métodos propios de esa clase.
- Es un código reutilizable

Herencia en JAVA

- Hay 3 palabras reservadas "nuevas" como son "**extends**", "**protected**" y "**super**".
  - → **extends**: Esta palabra reservada, indica a la clase hija cuál va a ser su clase padre Para que se entienda mejor, al poner esto estamos haciendo un "copy-paste dinámico" diciendo a la clase 'Futbolista' que se 'copie' todos los atributos y métodos públicos o protegidos de la clase 'SeleccionFutbol'. De aquí viene esa 'definición' que dimos de que la herencia en un 'copy-paste dinámico'.
  - → **protected**: sirve para indicar un tipo de visibilidad de los atributos y métodos de la clase padre y significa que cuando un atributo es 'protected' o protegido, solo es visible ese atributo o método desde una de las clases hijas y no desde otra clase.
  - → **super**: sirve para llamar al constructor de la clase padre.

Herencia en JAVA

```java
public class SeleccionFutbol {

        ......

        public SeleccionFutbol() {
        }

        public SeleccionFutbol(int id, String nombre, String apellidos, int edad) {
                this.id = id;
                this.Nombre = nombre;
                this.Apellidos = apellidos;
                this.Edad = edad;
```

```java
public class Futbolista extends SeleccionFutbol {
        ......
        public Futbolista() {
                super();
        }

        public Futbolista(int id, String nombre, String apellidos, int edad, int dorsal, String demarcacion) {
                super(id, nombre, apellidos, edad);
                this.dorsal = dorsal;
                this.demarcacion = demarcacion;
        }
```

Herencia en JAVA

```
public class Main {

	// ArrayList de objetos SeleccionFutbol. Idenpendientemente de la clase hija a la que pertenezca el objeto
	public static ArrayList<SeleccionFutbol> integrantes = new ArrayList<SeleccionFutbol>();

	public static void main(String[] args) {

		Entrenador delBosque = new Entrenador(1, "Vicente", "Del Bosque", 60, "284EZ89");
		Futbolista iniesta = new Futbolista(2, "Andres", "Iniesta", 29, 6, "Interior Derecho");
		Masajista raulMartinez = new Masajista(3, "Raúl", "Martinez", 41, "Licenciado en Fisioterapia", 18);

		integrantes.add(delBosque);
		integrantes.add(iniesta);
		integrantes.add(raulMartinez);

		// CONCENTRACION
		System.out.println("Todos los integrantes comienzan una concentracion. (Todos ejecutan el mismo método)");
		for (SeleccionFutbol integrante : integrantes) {
			System.out.print(integrante.getNombre()+" "+integrante.getApellidos()+" -> ");
			integrante.Concentrarse();
		}

		// VIAJE
		System.out.println("nTodos los integrantes viajan para jugar un partido. (Todos ejecutan el mismo método)");
		for (SeleccionFutbol integrante : integrantes) {
			System.out.print(integrante.getNombre()+" "+integrante.getApellidos()+" -> ");
			integrante.Viajar();
		}
	.....
```

Herencia en JAVA

- Vamos a ejecutar código específico de las clases hijas, de ahí que ahora no podamos recorrer el ArrayList y ejecutar el mismo método para todos los objetos ya que ahora esos objetos son únicos de la clases hijas.

```
// ENTRENAMIENTO
System.out.println("nEntrenamiento: Solamente el entrenador y el futbolista tiene metodos para entrenar:");
System.out.print(delBosque.getNombre()+" "+delBosque.getApellidos()+" -> ");
delBosque.dirigirEntrenamiento();
System.out.print(iniesta.getNombre()+" "+iniesta.getApellidos()+" -> ");
iniesta.entrenar();

// MASAJE
System.out.println("nMasaje: Solo el masajista tiene el método para dar un masaje:");
System.out.print(raulMartinez.getNombre()+" "+raulMartinez.getApellidos()+" -> ");
raulMartinez.darMasaje();

// PARTIDO DE FUTBOL
System.out.println("nPartido de Fútbol: Solamente el entrenador y el futbolista tiene metodos para el partido de fútbol:");
System.out.print(delBosque.getNombre()+" "+delBosque.getApellidos()+" -> ");
delBosque.dirigirPartido();
System.out.print(iniesta.getNombre()+" "+iniesta.getApellidos()+" -> ");
iniesta.jugarPartido();
```

Herencia en JAVA

- Se pide construir la clase Estudiante con los atributos nombre, edad y créditos matriculados.Se dispone de la clase Persona ya implementada.

```
class Persona{
     private String nom;
     private int edad;
     public Persona(String nom, int edad){ this.nom = nom; this.edad = edad; }
     public String getNom() { return this.nom; }
     public String toString() {return “Nombre: “+this.nom+” Edad:”+this.edad; }
```

- Opciones posibles:
  - → **Inapropiada**: Ignorar la clase Persona y construir la clase Estudiante con tres atributos (edad, nombre y créditos). Se repite la declaración de atributos y métodos ya hecha en la clase Persona.
  - → **Apropiada**: Utilizar la herencia para definir la clase Estudiante en base a la clase Persona.

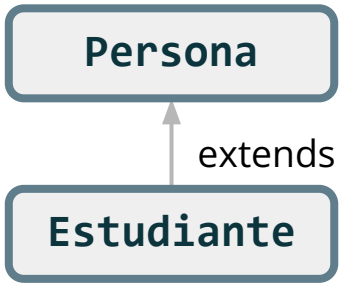
Herencia en JAVA

Construcción de la clase derivada Estudiante a partir de la clase base Persona.

- La clase Estudiante hereda (puede utilizar) todos los atributos y métodos que no son privados en Persona.

```
class Persona{
      private String nom;
      private int edad;
      public Persona(String nom, int edad){
      this.nom = nom; this.edad = edad; }
      public String toString() { return “Nombre: “+this.nom+” Edad:”+this.edad; }
      public String getNom() { return this.nom; }  }
```

```
class Estudiante extends Persona{
      private int creditos;
      public Estudiante(String nom, int edad)
      { super(nom,edad); this.creditos=60; }
      public int getCreditos() { return this.creditos; }
}
```



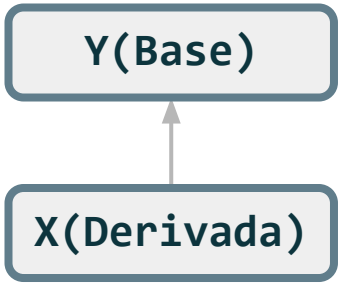
Herencia en JAVA

```
class TestEstudiantePersona {
public static void main (String[] args) {
    Estudiante e = new Estudiante(“Luís García”,20);
    Persona p = new Persona(“Luís García”, 20);
    System.out.println(“Persona: “ + p.toString());
    System.out.println(e.getNom()+” : ”+e.getCreditos()+”créditos”); }
}
```

- Se pueden invocar los métodos declarados en Persona desde un objeto Estudiante ya que Estudiante hereda de Persona.
  - La clase Estudiante puede acceder a los atributos y los métodos no privados de la clase Persona.

Herencia en JAVA

- Si X **Es UN(A)** Y,
  - Se dice que la clase derivada X es una variación de la clase base Y
  - Se dice que X e Y forman una jerarquía, donde la clase X es una **subclase** (o clase **derivada**) de Y e Y es una **superclase** (o clase **padre**) de X.
  - La relación es **transitiva**: si X ES UN(A) Y e Y ES UN(A) Z, entonces X ES UN(A) Z.
  - X puede referenciar todos los atributos y métodos que sean derivados en Y.
  - X es una clase completamente nueva e independiente (los cambios en X no afectan a Y).
- **Java no soporta la herencia múltiple**
  - Una clase solamente puede heredar como máximo de otra.



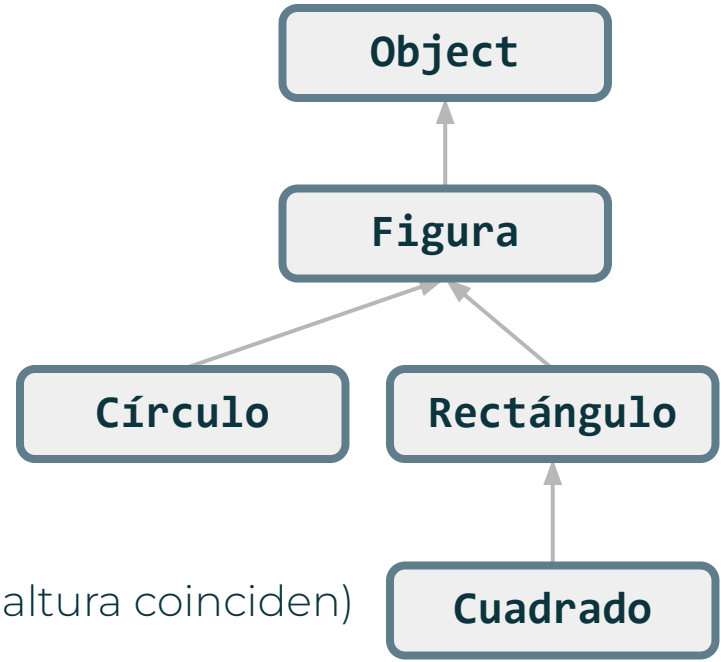
Herencia en JAVA

- Cualquier clase Java hereda implícitamente de la clase predefinida Object. Esta define los métodos que pueden ser invocados con cualquier objeto Java.

  Por ejemplo:

  - public String toString ()
  - public boolean equals (Object X)

- Ejemplo de la jerarquía de clases:
  - Una Figura ES UN Object
  - Un Círculo ES UNA Figura
  - Un Rectángulo ES UNA Figura
  - Un Cuadrado ES UN Rectángulo(la base y la altura coinciden)

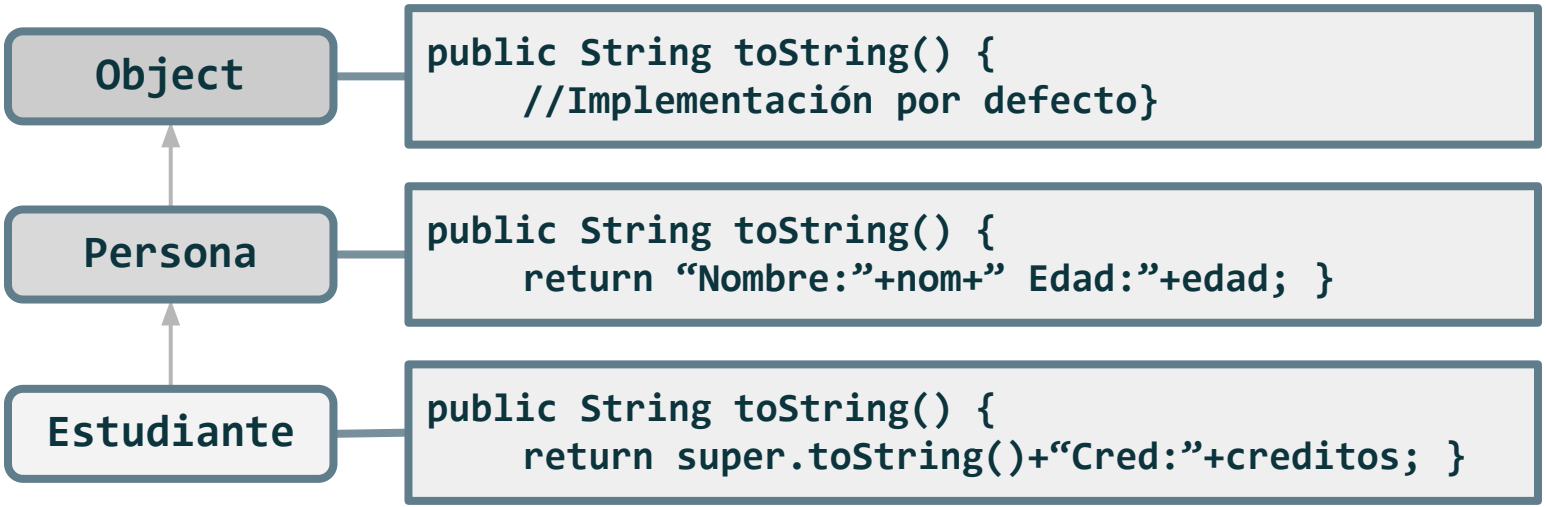


Sobreescritura de métodos

- Todo método no privado de la clase base que se defina de nuevo en la clase derivada se sobreescribe:
- Sobreescritura **completa**:
  - → Se define en la clase derivada un método:
    - Con el mismo perfil que en la base (nombre y lista de parámetros).
    - Con el mismo tipo de resultado que en la base.
- Sobreescritura parcial:
  - → Cuando solamente se quiere cambiar parcialmente el comportamiento del método de la clase base. Se utiliza **super** para invocar el método de la clase base

Sobreescritura de métodos

- Ejemplo:
  - Un ejemplo es la sobreescritura del método toString



  - La clase Persona sobreescribe **complementamente** el método. La clase Estudiante, sobreescribe **parcialmente** el método.

Sobreescritura de métodos

- Ejemplo:

```
class TestEstudiantePersona2 {
public static void main (String[] args) {
    Estudiante e = new Estudiante(“Luís García”,20);
    Persona p = new Persona(“Luís García”, 20);
    System.out.println(“Persona: “ + p);
    System.out.println(“Estudiante: “ + e); }
```

```
SALIDA POR PANTALLA
```

```
Persona: Nombre: Luís García Edad: 20
Estudiante: Nombre: Luís García Edad: 20 Cred: 60
```

  - No es necesario invocar explícitamente el método toString() del objeto, Java lo hace automáticamente para poder concatenar con un String.

Ejemplo documentación en JAVA

java.lang

Class Number

La clase Number hereda de Object (como todas)

java.lang.Object
    java.lang.Number

All Implemented Interfaces:

`Serializable`

Subclases directas de la clase Number

Direct Known Subclasses:

`AtomicInteger, AtomicLong, BigDecimal, BigInteger, Byte, Double, DoubleAccumulator, DoubleAdder, Float, Integer, Long, LongAccumulator, LongAdder, Short`

```
public abstract class Number
extends Object
implements Serializable
```

The abstract class `Number` is the superclass of platform classes representing numeric values that are convertible to the primitive types `byte`, `double`, `float`, `int`, conversion from the numeric value of a particular `Number` implementation to a given primitive type is defined by the `Number` implementation in question. For plat narrowing primitive conversion or a widening primitive conversion as defining in *The Java™ Language Specification* for converting between primitive types. The overall magnitude of a numeric value, may lose precision, and may even return a result of a different sign than the input. See the documentation of a given `Numbe`

Since:

`JDK1.0`

Cabecera de la clase Number

See Also:

`Serialized Form`

See *The Java™ Language Specification*:

`5.1.2 Widening Primitive Conversions, 5.1.3 Narrowing Primitive Conversions`

Constructor Summary

Constructor

Constructors

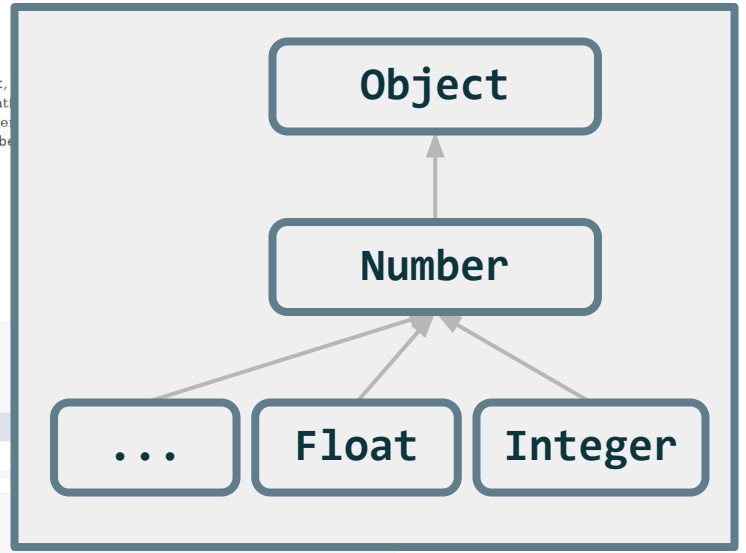
Constructor and Description
`Number()`

Method Summary

Métodos

All Methods | **Instance Methods** | **Abstract Methods** | **Concrete Methods**

Modifier and Type	Method and Description
byte	`byteValue()`
	Returns the value of the specified number as a `byte`, which may involve rounding or truncation.



Interfaces

Clases abstractas

- Las utilizaremos cuando necesitemos crear una clase padre donde únicamente coloquemos una estructura muy general, dejando que sean las clases hijas quienes definan los detalles.
- Una clase abstracta es como una clase convencional, con la diferencia que debe poseer por lo menos un método abstracto.
- Un método abstracto es un método vacío: un método el cual no posee cuerpo, por ende no puede realizar ninguna acción.
- La utilidad de un método abstracto es definir qué se debe hacer pero no el cómo se debe hacer.

Clases abstractas

- Veamos un ejemplo para que nos quede más en claro:

```
public class Figura {
        private int numeroLados;
        public Figura() {
            this.numeroLados = 0;
        }
        public float area() {
            return 0f;
        }
}
```

- En este caso la clase posee una atributo, un constructor y un método, a partir de esta clase podré generar la n cantidad de figuras que necesite, ya sean cuadrados, rectangulos, triangulos, circulos etc...

Clases abstractas

- Dentro de la clase encontramos el método área, método que se encuentra pensado para obtener el área de cualquier figura, sin embargo cómo sabemos todas las figuras poseen su propia fórmula matemática para calcular su área.
- Si yo comienzo a heredar de la clase Figura todas las clases hijas tendrían que sobre escribir el método área e implementar su propia fórmula para así poder calcular su área.
- En estos casos, en los casos la clase hija siempre deba que sobreescribir el método lo que podemos hacer es convertir al método convencional en un método abstracto, un método que defina qué hacer, pero no cómo se deba hacer.

Clases abstractas

- Por tanto la clase Figura quedaría de la siguiente forma:

```
public abstract class Figura {

    private int numeroLados;
    public Figura()
    {
        this.numeroLados = 0;
    }
    public abstract float area();
}
```

Clases abstractas

- Es importante mencionar que las clases abstractas pueden ser heredadas por la cantidad de clases que necesitemos, pero no pueden ser instanciadas.
- Para heredar de una clase abstracta basta con utilizar la palabra reservada extends.

```
public class Triangulo extends Figura { ..... }
```

- Al nosotros heredar de una clase abstracta es obligatorio implementar todos sus métodos abstractos, es decir debemos definir comportamiento, definir cómo se va a realizar la tarea.

Interfaces

- La POO por medio de la herencia nos permite crear nuevas clases partiendo (o extendiendo) de otras.
- No obstante, Java no admite herencia múltiple. Una clase solo puede extender a otra.
- Para poder ofrecer una funcionalidad muy similar a lo que sería la herencia múltiple java dispone de interfaces.
- Podemos definir a una **interface** como una colección de métodos abstractos y propiedades constantes en las que se especifica que se debe de hacer pero no como, serán las clases hijas quienes definan el comportamiento.
- Tiene las siguientes características:
  - → **Todos los miembros son públicos** (no hay necesidad de declararlos públicos)
  - → **Todos los métodos son abstractos** (A partir de JDK 8, hay métodos default))
  - → **Todos los campos datos son static y final**. Se usa para definir valores constantes.

Interfaces

- Para crear una Interface

```
modificador interface nombreInterface{
    variables;
    métodos;
}
```

- Como indicar que una clase implementa una interface:

```
modificador Class nombreClase implements nombreInterface{
    implementación métodos;
}
```

Si una clase implementa una interface deberá sobreescribir todos sus métodos

Interfaces

- Veamos un ejemplo:

```
public interface Canino {
    public abstract void aullar();
    public abstract void ladrar();
}
```

- Cómo podemos observar en la interfaz solo encontraremos métodos abstractos, método vacíos. Para poder implementar la interfaz basta con utilizar la palabra reservada implements.

```
public class Perro implements Canino { ... }
```

En esencia las interfaces serán contratos que indicarán que es lo que se debe hacer sin proveer ninguna funcionalidad.

Interfaces

- Veamos un ejemplo:

```
public interface Forma {
    void dibujar();
    double area();
}
```

```
public class Rectangulo implements Forma {
    int ancho, largo;
    public Rectangulo(int ancho, int largo) {
        this.ancho = ancho;
        this.largo = largo; }
    @override
    public void dibujar() { System.out.println("Rectángulo dibujado"); }
    @override
    public double area() { return ancho*largo; }
```

Interfaces

- Veamos un ejemplo:

```
public interface Forma {
    void dibujar();
    double area();
}
```

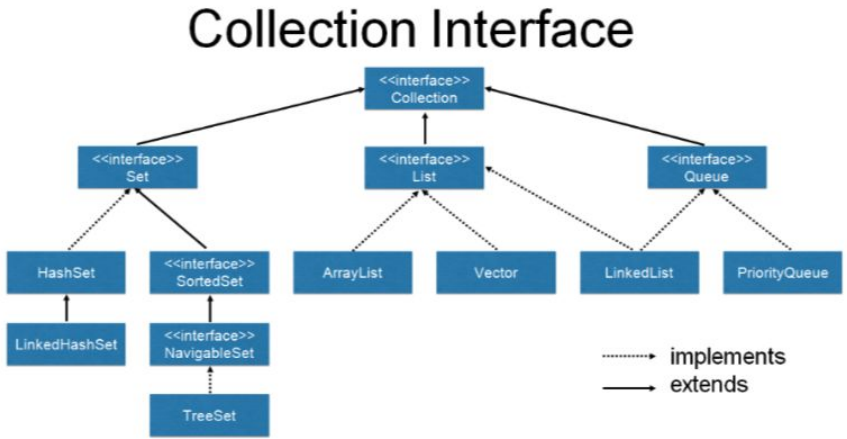
```
public class Circulo implements Forma {
    double pi = 3.14; int radio
    public Circulo(int radio) { this.radio = radio; }

    @override
    public void dibujar() { System.out.println("Círculo dibujado"); }

    @override
    public double area() { return (double)((pi*radio*radio)/2); }
```

Interface vs Clase abstracta

- En una interface los atributos deben ser constantes. En una clase abstracta pueden ser de cualquier tipo
- Una clase abstracta puede contener métodos que dispongan de implementación. Una interface sólo dispone de métodos abstractos.
- Como todos los métodos de una interface son abstractos no necesitamos incluir abstract en la definición de estos. En cambio, en una clase abstracta sí que necesitaremos hacerlo en aquellos métodos que lo sean.
- Una clase abstracta no es más que una clase común la cual posee atributos, métodos, constructores y por lo menos un método abstracto. Una clase abstracta no puede ser instanciada, solo heredada.
- Cómo Java no permite la herencia múltiple habrá ocasiones en las cuales debamos utilizar interfaces, las cuales podemos verlas como contratos, contratos donde está muy bien establecido que debe hacer la clase que la implementa.

Iterador

- De forma genérica un iterador es un objeto que podemos usar para obtener todos los objetos de una colección uno a uno.
- En Java, un Iterator es una interfaz que puede ser implementada por una clase de colección.
- Cualquier colección puede crear un objeto de tipo Iterator. Este le proveerá una forma fácil de obtener uno a uno todos los objetos que posea.



Iterador

- Principales métodos:
  - **next()**: retorna un objeto de tipo Object empezando por el primero y establece el iterator para que retorne el próximo objeto en la siguiente llamada a este mismo método. Si no existe próximo objeto y la función next() es llamada se producirá una NoSuchElementException.
  - **hasNext()**: retorna true si existe un próximo objeto a retornar a través de la llamada a la función next().
  - **remove()**: Elimina el último objeto retornado por la función next(). Si no se invoca next() antes de remove() o se invoca dos veces después de next(), se produce una IllegalStateException.

Iterador

- Ejemplo:

```
public static void main (String args[]) {
     ArrayList<String> locations = new ArrayList<String>();
     locations.add(“New York”);
     locations.add(“Tokyo”);
     locations.add(“París”);
     System.out.println(“Lista de ciudades”);
     Iterator it = locations.iterator();
     it.next();
     it.remove();
     while(it.hasNext())
     {
          Object abc = it.next();
          System.out.println(abc);
     }
     System.out.println(“ “);
}
```

Tipos enumerados

- Es un tipo de clase Java:

```
public enum TiposDeTelevisores { ...   }
```

- Características:
  - → El compilador agrega automáticamente el método values() que devuelve un array con todos los valores del enum en el orden en que son declarados.
  - → Se usa con for-each para iterar sobre los valores de un tipo enum.
  - → Un tipo enumerado puede añadir campos constantes al objeto enumerado y recuperar esos campos.
  - → No podemos crear objetos del tipo enumerado con la sentencia new

```
public enum TiposDeTelevisores { SAMSUNG, LG, SONY  }
public static void main(){
     System.out.println(TiposTelevisores.SAMSUNG);
}
```

Excepciones

Excepciones

- Si en la ejecución de una pieza de código se prevé que pueda ocurrir un error, hay dos formas de tratar ese posible error:
  - → Escribir el código para evitar que el error ocurra (método utilizado hasta ahora)

```
if(divisor != 0) cociente = dividendo/divisor;
     else System.out.println("División entre cero");
```

  - → Escribir código que maneje el error cuando éste ocurra. Esto requiere que el lenguaje provea de un mecanismo que detecte el error y lo notifique.

- **Java posee un mecanismo de detección y notificación de errores.**

Concepto de excepción

- Una **excepción** en Java es un error o situación excepcional que se produce durante la ejecución de un programa. Algunos ejemplos de errores y situaciones excepcionales son:
  - → Leer un fichero que no existe
  - → Acceder al valor N de una colección que contiene menos de N elementos
  - → Enviar/recibir información por red mientras se produce una pérdida de conectividad
- Este mecanismo permite tratar los errores de una forma elegante ya que **separa el código para el tratamiento de errores del código normal del programa**.
- Todas las excepciones en **Java** se representan, como vamos a ver en la siguiente sección, a través de **objetos** que **heredan**, en última instancia, de la clase **java.lang.Throwable**.

La jerarquía Throwable

Lista de algunas de las excepciones más importantes:

1. **ArithmeticException** Se produce con las operaciones aritméticas.
2. **ArrayIndexOutOfBoundsException** Se produce cuando se intenta acceder a una posición de un array que no existe.
3. **ClassNotFoundException** Esta excepción se genera cuando intentamos acceder a una clase cuya definición no se encuentra
4. **FileNotFoundException** Esta excepción se genera cuando un archivo no es accesible o no se abre.
5. **IOException** Se produce cuando una operación de entrada-salida falla o se interrumpe
6. **NoSuchFieldException** Se genera cuando una clase no contiene el atributo especificado
7. **NoSuchMethodException** Se genera al acceder a un método que no se encuentra.
8. **NullPointerException** Esta excepción se genera cuando se hace referencia a un objeto nulo.
9. **NumberFormatException** Esta excepción se produce cuando un método no puede convertir una cadena en un formato numérico.
10. **RuntimeException** Esto representa cualquier excepción que ocurra durante el tiempo de ejecución.
11. **InputMismatchException** El valor de entrada está fuera de un determinado rango

Capturar excepciones

¿Qué hacemos con una excepción una vez se ha producido?

- Podemos lanzarlas, propagarlas y capturarlas.
  - → Para **lanzarlas** usaremos **throw**
  - → Para **propagarlas** usaremos **throws**
  - → Para **capturarlas** usaremos **try-catch**

Capturar excepciones

Para capturar un posible error mediante excepciones se utilizan las siguientes sentencias:

- **try**: contiene las instrucciones donde se producen uno o más tipos de excepciones:
  - Si se produce una excepción, el flujo de ejecución salta directamente al catch de esta excepción.
- **catch**: contiene las instrucciones a realizar si se produce una excepción:
  - Indica que tipo de excepción se captura
  - Si hubiese más de una excepción posible, habría más de un bloque catch.
- **finally**: contiene instrucciones que se ejecutan tanto si se completa el try como si no:
  - Es opcional, se utiliza por si se producen excepciones que no son comprobadas ni capturadas y que abortarían el programa pero que, aún así, hay que hacer determinadas acciones antes de terminar. Por ejemplo, guardar los datos y cerrar un fichero en disco

Capturar excepción

- Ejemplo 1:

```
//bloque1
try {
    //bloque2
} catch (Exception error) {
    //bloque3
}
//bloque4
```

- Sin excepciones: **1 → 2 → 4**
- Con una excepción en el bloque 2: **1 → 2* → 3 → 4**
- Con una excepción en el bloque 1: **error**

Capturar excepción

- Ejemplo 2:

```
//bloque1
try {
    //bloque2
} catch (AritmeticException error) {
    //bloque3
} catch (NullPointerException error) {
    //bloque4
}
//bloque5
```

- Sin excepciones: **1 → 2 → 5**
- Con una excepción de tipo aritmético: **1 → 2* → 3 → 5**
- Con una excepción de acceso a un objeto nulo: **1 → 2* → 4 → 5**
- Excepción de otro tipo: **1 → 2***

Capturar excepción

- Ejemplo 3:

```
//bloque1
try {
    //bloque2
} catch (AritmeticException error) {
    //bloque3
} finally {
    //bloque4
}
//bloque5
```

- Sin excepciones: **1 → 2 → 4 → 5**
- Con una excepción de tipo aritmético: **1 → 2* → 3 → 4 → 5**
- Excepción de otro tipo: **1 → 2* → 4**

Capturar excepciones

Catch

- Podemos especificar tantos como tipos de excepciones queramos gestionar
- O bien, podemos capturar cualquier especificando **Exception**, ya que es la superclase común de todas las excepciones.
- Métodos de la clase Exception:
  - **getMessage** obtenemos una cadena descriptiva del error
  - **printStackTrace** se muestra por la salida estándar la traza de errores que se han producido

```
//bloque1
try {
     ... //Aquí va el código que puede lanzar una excepción
} catch (Exception e) {
     System.out.println("El error es: " + e.getMessage())
     e.printStackTrace();
}
```

Capturar excepciones

Finally

- Si el cuerpo del bloque try llega a comenzar su ejecución, el bloque **finally** siempre se ejecutará...
  - Detrás del bloque try si no se producen excepciones
  - Después de un bloque catch si éste captura la excepción
  - Justo después de que se produzca la excepción si ninguna cláusula catch captura la excepción y antes de que la excepción se propague hacia arriba

```
public static void main (String args[]) {
     try { int a = 30, b = 0;
           int c = a / b ; //división entre cero
           System.out.println(a + "/" + b + "=" + c);
     } catch (ArithmeticException e) {
          System.out.println("No se puede dividir un número entre
     cero");
     }
}
```

Capturar excepciones

Ejemplo:

```
public static void main (String args[]) {

     try {
          int num = Integer.parseInt(“hola”); //hola no se puede convertir a int
          System.out.println(num);
     } catch (NumberFormatException e) {
          System.out.println(“El formato del número es incorrecto”);
     }
}
```

Capturar excepciones

Ejemplo:

```
public static void main (String args[]) {
     Scanner entrada = new Scanner(System.in);
     int decimal, resultado;
     try {
          decimal = entrada.nextInt();
          resultado = 100 / decimal;
     } catch (Exception e) {
          System.out.println("Error: " + e.getMessage());
          e.printStackTrace();
     }
     System.out.println("El programa imprime esta frase con normalidad");
}
```

Lanzar una excepción

Se utiliza la sentencia **throw** para lanzar objetos de tipo Throwable

```
throw new Exception(“Mensaje de error...”);
```

- Cuando se lanza una excepción :
  - Se sale inmediatamente del bloque de código actual
  - Si el bloque tiene asociada una cláusula catch adecuada para el tipo de excepción generada, se ejecuta el cuerpo de la cláusula catch
  - Si no, se sale inmediatamente del bloque o método dentro del cual está el bloque en el que se produjo la excepción y se busca una cláusula catch apropiada.
  - El proceso continúa hasta llegar al método main de la aplicación. Si ahí tampoco existe una cláusula catch adecuada, la JVM finaliza su ejecución con un mensaje de error.

Lanzar una excepción

Ejemplo:

```
public static void main (String args[]) {

     Scanner entrada = new Scanner(System.in);
     int minutos;

     System.out.println("Indica los minutos");
     minutos = entrada.nextInt();

     if(minutos < 0 || minutos >= 60 {
     throw new InputMismatchException("Valor fuera de rango, de 0 a 60");
     }

}
```

Lanzar una excepción

Ejemplo:

```
static void fun() throws NullPointerException {
     try {
          throw new NullPointerException("demo");
     } catch (NullPointerException e) {
          System.out.println("Capturando excepción dentro de un método");
     }
}
public static void main (String args[]) {
     try {
          fun();
     } catch (NullPointerException e) {
          System.out.println("Capturando excepción en el main");
     }
}
```

Propagar una excepción

Si en el cuerpo de un método se lanza una excepción (de un tipo derivado de la clase Exception), en la cabecera del método hay que añadir una cláusula throws junto con la lista de tipos de excepciones que se pueden producir al invocar el método.

Ejemplo:

```
public String leerFichero (String nombreFichero) throws IOException
{ … }
```

Por tanto, al implementar un método hay que decidir si las excepciones se propagarán hacia arriba (throws) o se capturan en el propio método(catch).

Un método que propaga la excepción

```
public void f() throws IOException
{
    //Fragmento de código que puede
    //lanzar una excepción de tipo IOException
}
```

Propagar una excepción

Un método equivalente al anterior que NO propaga la excepción

```
public void f()
{
     try {
          //Fragmento de código que puede
          //lanzar una excepción de tipo IOException
     } catch (IOException e) {
          //Tratamiento de la excepción
     } finally {
          //Liberar recursos, siempre se hace
     }
}
```

Propagar una excepción

Ejemplo:

```
public static void ejemplo(int x, int[] v) throws ArrayIndexOutOfBoundsException {
      if( x < 0 || x>= v.length ) { throw new ArrayIndexOutOfBoundsException (“Posición no válida.”); }
      System.out.println(v[x]);
}
public static void main (String args[]) {
      Scanner entrada = new Scanner(System.in);
      boolean salir = false;
      int pos;
      int[] elArray = {1, 2, 3};
      do {  System.out.println(“Dime una posición para saber su valor”);
            pos = entrada.nextInt();
            try {
                  ejemplo(pos, elArray);
                  salir = true;
            } catch (Exception ArrayIndexOutOfBoundsException e) {
                  System.out.println(“Se ha producido una excepción: ” + e );
            }
      }while(!salir);
}
```

Excepciones de usuario

También existe la posibilidad de definir **excepciones de usuario** para representar errores de lógica de la aplicación, que el programador quiera gestionar.

1. Se debe crear una clase que herede de **Exception**
2. Se añaden los **métodos** y **propiedades** para almacenar información relativa a nuestro tipo de error.
3. Para utilizar estas excepciones (capturarlas, lanzarlas o propagarlas) hacemos **lo mismo** que lo explicado antes para las excepciones predefinidas de **Java**.

Excepciones de usuario

Ejemplo:

- Definimos una subclase de un tipo de excepción ya existente

```
public class ClienteExisteException extends Exception
{
     private Cliente cliente;
     public ClienteExisteException (Cliente cliente) {
          this.cliente=cliente;
     }
     @override
     public String toString() {
          return “El cliente de nombre “ + this.cliente.getNombre() + “ ya existe.”);
     }
}
```

Excepciones de usuario

Ejemplo:

Adaptamos los métodos para que se lance la excepción:

```
public static void nuevoCliente(String nombre, ArrayList<Cliente> clientes)
throws ClienteExisteException
{
      Cliente nuevoCliente = buscarCliente(nombre, clientes);

      if(nuevoCliente != null )
      {  throw new ClienteExisteException(nuevoCliente); }
      else {
                  nuevoCliente = new Cliente(nombre);
                  clientes.add(nuevoCliente);
                  System.out.println("Cliente dado de alta correctamente.");
            }
}
```

Al invocar al método debemos capturar la excepción.

```
try { nuevoCliente(nombre, clientes); }
catch (ClienteExisteException e) { System.out.println("Error: " + e); }
```

En resumen...

- Con la cláusula throws se puede propagar la excepción por toda la pila de llamadas, situando el tratamiento donde se quiera.
- Puede haber más de una excepción (separadas por comas) en una misma cláusula throws.
- No confundir la cláusula throws para propagar la excepción con throw para lanzar una nueva excepción (Recordad que hay que tenerla creada, ya que es un objeto).
- Al igual que con el resto de estructuras de control, las instrucciones try-catch- finally pueden situarse dentro de bucles, condicionales, etc.

Preguntas