IES DR LLUÍS SIMARRO

ENSENYAMENT PÚBLIC

XÀTIVA

CFGS Desarrollo de aplicaciones web

Módulo profesional: Programación

GENERALITAT VALENCIANA

Conselleria d'Educació, Investigació, Cultura i Esport

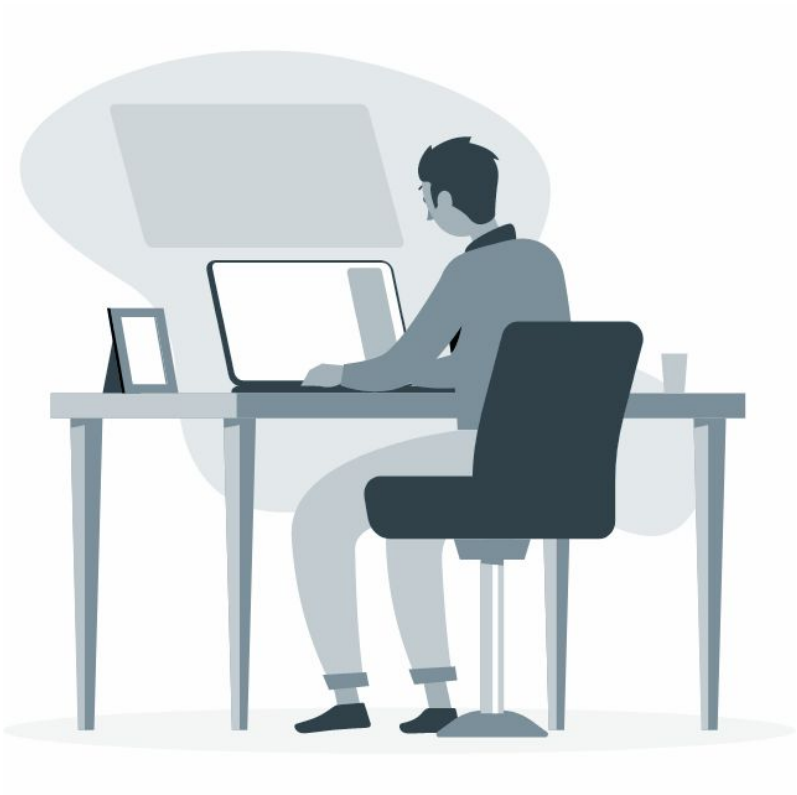
GOBIERNO DE ESPAÑA

MINISTERIO DE EDUCACIÓN Y FORMACIÓN PROFESIONAL

Unió Europea

Fons Social Europeu

L'FSE inverteix en el teu futur



Material elaborado por:

Edu Torregrosa Llácer

(aulaenlanube.com)

Herencia y excepciones en JAVA

1. Herencia en JAVA
    a. Estructura de clases
    b. Sobreescritura de métodos
    c. Polimorfismo
    d. Interfaces
    e. Clases abstractas
2. Comparar objetos
3. Tratamiento de excepciones
    a. Capturar excepciones
    b. Lanzar excepciones
    c. Propagar excepciones
    d. Excepciones personalizadas

Introducción

La **Herencia** es uno de los 4 pilares de la programación orientada a objetos junto con la Abstracción, Encapsulación y Polimorfismo.

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, es decir, de qué clase va a heredar tanto sus atributos como sus métodos. Una clase en JAVA puede extender a una única clase. Sin embargo, podemos tener todos los niveles de Herencia que necesitemos
- **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**: se utiliza para hacer referencia a la clase padre o a la superclase. Nos permite en la definición de una clase hija, acceder a los atributos y métodos de la clase padre. El uso más común es para llamar a un constructor de la clase padre desde el constructor de la clase hija, en ese caso la llamada deberá ser la primera instrucción del método.

Introducción

```
public class Padre {
     String nombre;
     public Padre(String nombre) {
        this.nombre = nombre;
        System.out.print(nombre);
     }
}

public class Hija extends Padre {
     int edad;
     public Hija(int edad, String padre) {
       super(padre);
       this.edad = edad;
       System.out.println(" tiene una hija de " + edad + " años");
     }
}
```

Introducción

```
public class Padre {
     String nombre;
     public Padre(String nombre) {
        this.nombre = nombre;
        System.out.print(nombre);
     }
     public void mostrarInfo() {  System.out.println("Padre: " + nombre);  }
}
public class Hija extends Padre {
     String nombre;
     int edad;
     public Hija(int edad, String padre, String hija) {
       super(padre);
       this.nombre = hija;
       this.edad = edad;
       System.out.println(" tiene una hija de " + edad + " años");
     }
     public void mostrarInfo() {
        System.out.println("Hija: " + nombre + ", edad: " + edad + " años");
        super.mostrarInfo();
     }
}
```

Ejemplo Herencia en JAVA : Clase Persona

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.

```
public class Persona{
     private String nombre;
     private int edad;
     public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
     }
     public String getNombre() {  return this.nombre; }
     public String toString()  {  return "Nombre: " + this.nombre + " 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.

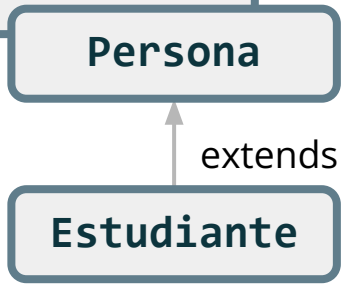
Herencia en JAVA

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.

```
public class Persona {
      private String nombre;
      private int edad;
      public Persona(String nombre, int edad) {
         this.nombre = nombre;
         this.edad = edad;
      }
      public String toString()  { return "Nombre: " + this.nombre + " Edad:" + this.edad; }
      public String getNombre() { return this.nombre; }
}
```

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



Herencia en JAVA

```
public class AppEstudiantePersona {
    public static void main (String[] args) {
        Estudiante e = new Estudiante("Luís García",20);
        Persona p = new Persona("Marta Gómez", 20);
        System.out.println(p.getNombre());
        System.out.println(e.getNombre()+" : "+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.

p.getNombre( ) y **e.getNombre( )** están accediendo al mismo método, pero lo hacen desde instancias de objetos distintos(Persona y Estudiante)

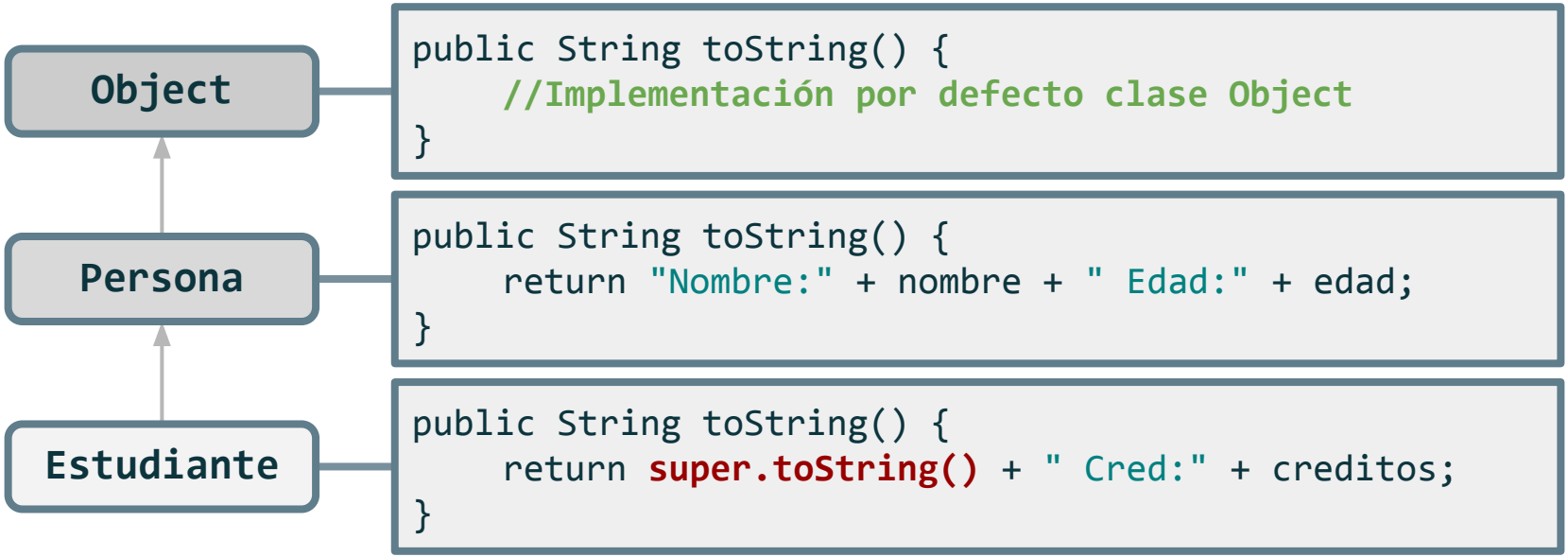
Sobreescritura de métodos en JAVA

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.

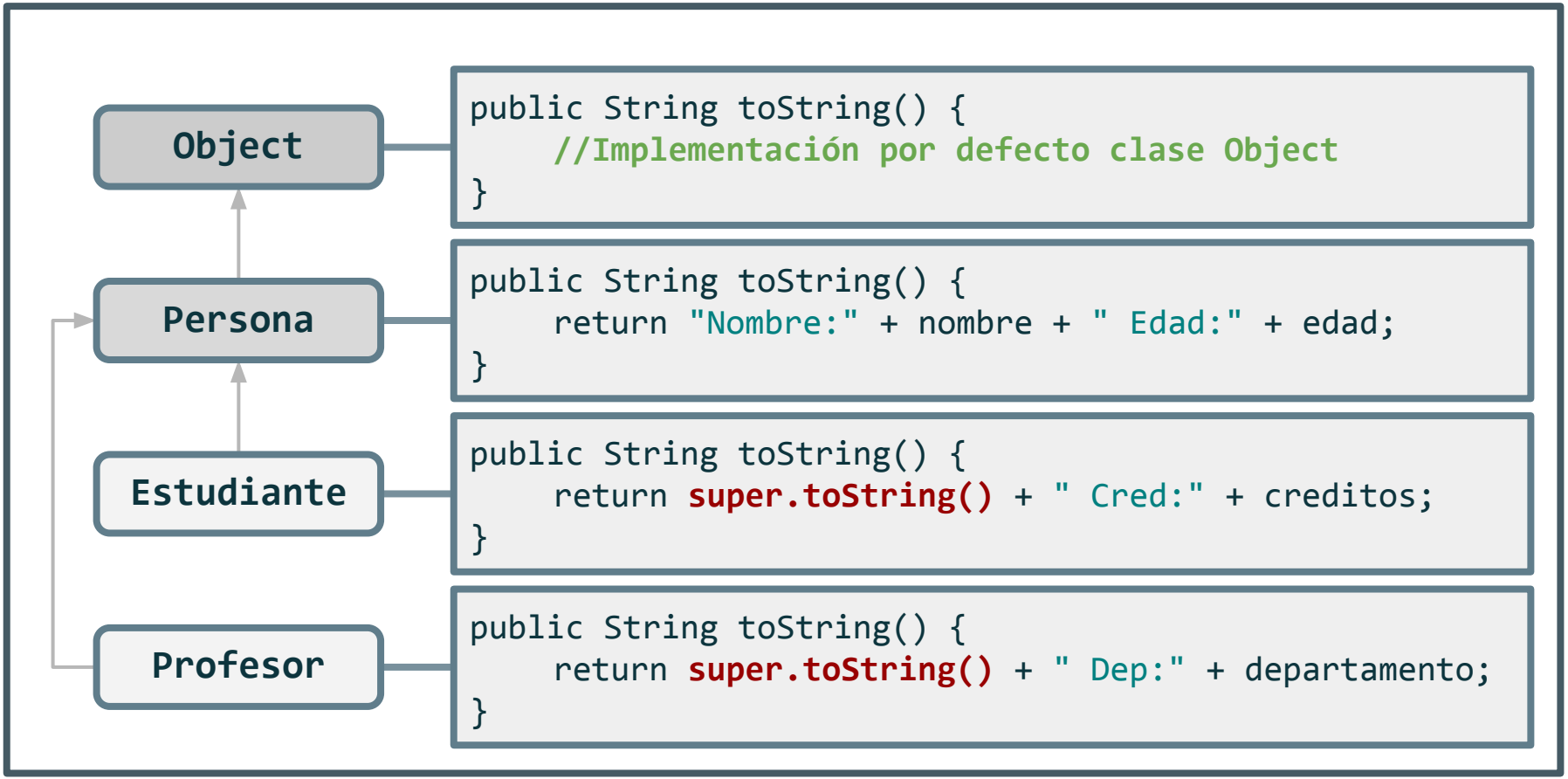
Sobreescritura de métodos

- 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



Sobreescritura de métodos

```
class AppEstudiantePersona {
     public static void main (String[] args) {
          Persona p = new Persona("Luís García", 20);
          Estudiante e = new Estudiante("Luís García",20);
          Profesor pr = new Profesor("Luís García", 20, "Informática");
          System.out.println(p);
          System.out.println(p.toString());
          System.out.println(e);
          System.out.println(e.toString());
          System.out.println(pr);
          System.out.println(pr.toString());
     }
}
```

SALIDA POR PANTALLA

```
Nombre: Luís García Edad: 20
Nombre: Luís García Edad: 20
Nombre: Luís García Edad: 20 Cred: 60
Nombre: Luís García Edad: 20 Cred: 60
Nombre: Luís García Edad: 20 Dep: Informática
Nombre: Luís García Edad: 20 Dep: Informática
```

Polimorfismo en JAVA

Polimorfismo en JAVA

El polimorfismo permite a las clases hijas heredar y redefinir los métodos de sus clases padres y tratar a los objetos de esas clases de manera genérica, sin importar su tipo específico. Hay dos formas principales de polimorfismo en JAVA: el polimorfismo de sobrecarga y el polimorfismo de sobrescritura.

- El **polimorfismo de sobrecarga** se refiere a la capacidad de una clase para tener métodos con el mismo nombre, pero con diferentes argumentos.
- El **polimorfismo de sobrescritura** se refiere a la capacidad de una clase hija para redefinir o sobreescribir los métodos de su clase padre.

```
public class Ejemplo {
   void suma(int a, int b) {
      System.out.println("a+b="+(a + b));
   }
   void suma(double a, double b) {
      System.out.println("a+b="+(a + b));
   }
   void suma(int a, int b, int c) {
      System.out.println("a+b+c="+(a + b + c));
   }
}
```

SOBRECARGA

```
public class Padre {
   void mensaje() {
      System.out.println("Padre");
   }
}
public class Hija extends Padre {
   void mensaje() {
      System.out.println("Hija");
   }
}
```

SOBRESCRITURA

Polimorfismo en JAVA

Tenemos una jerarquía de clases de 3 niveles, y en cada uno de ellos se redefine por completo el método **hacerRuido( )**.

En este ejemplo, la clase "Animal" es la superclase principal, y las clases "Perro" es la superclase de la clase "Pitbull". A continuación se muestra un ejemplo de polimorfismo de 3 niveles.

```
class Animal {
    public void hacerRuido() {
        System.out.println("El animal hace un ruido");
    }
}
class Perro extends Animal {
    public void hacerRuido() {
        System.out.println("El perro ladra");
    }
}
class Pitbull extends Perro {
    public void hacerRuido() {
        System.out.println("El pitbull tiene un ladrido profundo");
    }
}
```

```
public static void main(String[] args) {
     ArrayList<Animal> animales = new ArrayList<>();
     animales.add(new Animal());
     animales.add(new Perro());
     animales.add(new Pitbull());
     for (Animal a : animales) a.hacerRuido();
}
```

Clase Object en JAVA

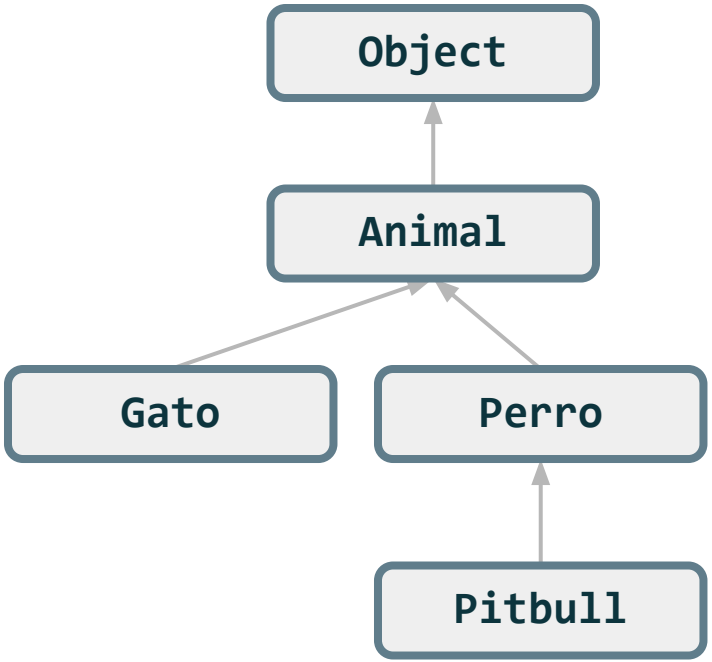
Clase Object en JAVA

- **Toda clase en JAVA hereda implícitamente de la clase Object**. Ésta define los métodos que pueden ser invocados con cualquier objeto de JAVA.

  Por ejemplo:

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

- Ejemplo de la jerarquía de clases:
  - Un Animal es un Object
  - Un Perro es un Animal y un Object
  - Un Gato es un Animal y un Object
  - Un Pitbull es un Perro, un Animal y un Object



Clase Object en JAVA

- La clase Object es una superclase predeterminada de todas las clases. Todas las clases de Java directa o indirectamente heredan de la clase Object.
- La clase Object define una serie de métodos que están disponibles en todas las clases en JAVA. Podemos destacar los siguientes métodos:
    - **clone( )**: Crea y devuelve una copia del objeto actual.
    - **equals(Object obj)**: Compara el objeto actual con el objeto dado y devuelve verdadero si ambos objetos son iguales.
    - **getClass( )**: Devuelve la clase del objeto actual.
    - **hashCode( )**: Devuelve un código hash único para el objeto actual.
    - **toString( )**: Devuelve una representación de cadena del objeto actual.
- Además, como ya hemos visto, todas las clases en JAVA pueden sobrescribir los métodos de la clase Object. Por ejemplo, una clase puede sobrescribir el método toString() para proporcionar una representación de cadena personalizada del objeto.

Clase Object en JAVA

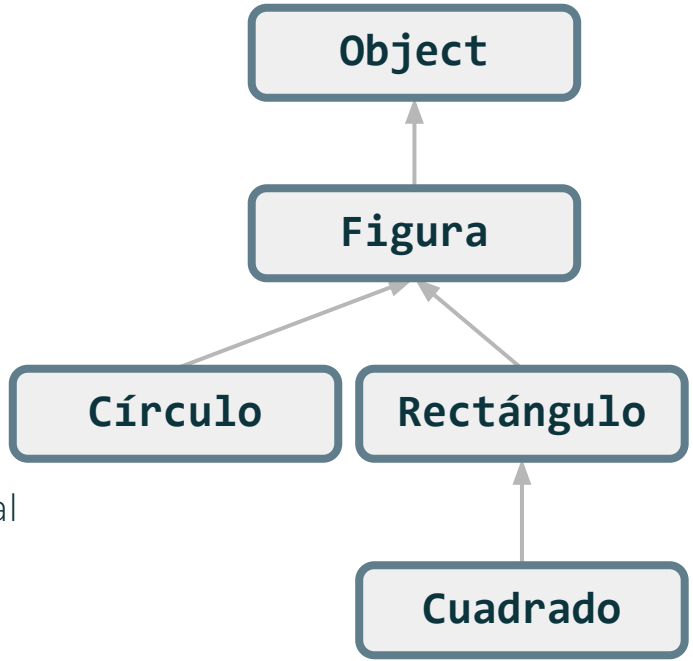
```java
public class Persona {
    private String nombre;
    private int edad;

    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }
    public String toString() {
        return "Nombre: " + nombre + ", Edad: " + edad;
    }
}
```

```java
public class Punto {

    private int x;
    private int y;

    public Punto(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public boolean equals(Object obj) {
        if (obj == null) return false;
        if (obj == this) return true;
        if (!(obj instanceof Punto)) return false;
        Punto otroPunto = (Punto) obj;
        return this.x == otroPunto.x && this.y == otroPunto.y;
    }
}
```

Clases abstractas en JAVA

Herencia en JAVA

- **Una clase abstracta es una clase que no se puede instanciar directamente**, y se utiliza como un modelo para crear otras clases que la extienden. Es decir, una clase abstracta sirve como una plantilla que define ciertos métodos y propiedades que se deben implementar en las clases que la extienden. Para que una clase sea abstracta, debe poseer por lo menos un método abstracto.
- 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.
- 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 claro. En este caso la clase posee una atributo, un constructor y un método, a partir de esta clase podremos generar todo tipo de figuras, ya sean cuadrados, rectángulos, triángulos, círculos etc...

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

- Dentro de la clase encontramos el método área, método que se encuentra pensado para obtener el área de cualquier figura, sin embargo, cada figura posee su propia fórmula matemática para calcular su área. Por ello, todas las clases hijas tendrían que sobrescribir el método área e implementar su propia fórmula para así poder calcular su área.

- En estos casos, cuando la clase hija siempre debe sobrescribir el método, lo que podemos hacer es convertir dicho método 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 2D {
    private int numeroLados;
    public Figura2D(int numeroLados) {  this.numeroLados = numeroLados;  }
    public abstract double area();
}
```

- Para heredar de una clase abstracta se hace del mismo modo que con clases no abstractas, basta con utilizar la palabra reservada extends.
- Al heredar de una clase abstracta es obligatorio implementar todos sus métodos abstractos, es decir debemos definir cómo se va a realizar la tarea.
- Desde una clase externa no podemos llamar al constructor de una clase abstracta. Sin embargo, las clases hijas pueden llamar al constructor de la clase abstracta a través de **super( )**.

Clases abstractas

```
public class Triangulo extends Figura2D {
     private Punto2D p1,p2,p3;
     public Triangulo(Punto2D p1, Punto2D p2, Punto2D p3) {
        super(3);
        this.p1 = p1;
        this.p2 = p2;
        this.p3 = p3;
     }
     public double area() {
        double a = Punto2D.distancia(p1, p2);
        double b = Punto2D.distancia(p2, p3);
        double c = Punto2D.distancia(p3, p1);
        double s = (a + b + c) / 2;
        return (Math.sqrt(s * (s - a) * (s - b) * (s - c)));
    }
}
```

```
public class Punto2D {
    private double x, y;
    public Punto2D(double x, double y) {
        this.x = x;
        this.y = y;
    }
    public static double distancia(Punto2D p1, Punto2D p2) {
        return Math.sqrt(Math.pow(p2.x - p1.x, 2) + Math.pow(p2.y - p1.y, 2));
    }
}
```

Ejercicios Clase Triángulo

A partir de las clases anteriores, Figura2D, Triángulo y Punto2D. Ampliar la clase triángulo para que incluya los métodos:

- **perimetro()**, deberá obtener el perímetro del triángulo, dicho perímetro se puede obtener a través de la suma de cada uno de los lados del triángulo.
- **esTriangulo()**, deberá comprobar si es un triángulo válido.
- **tipoTriangulo()**, deberá devolver el tipo de triángulo según sus lados. Los posibles valores serán NO_TRIANGULO, EQUILÁTERO, ISÓSCELES y ESCALENO.
- **toString()**, deberá mostrar toda la información del triángulo, coordenadas de los puntos, tipo, área y perímetro. Además se deberá redondear tanto el área como el perímetro mostrando únicamente 1 decimal. A continuación se muestra un ejemplo de salida del método.

```
El Triángulo es de tipo isósceles, de área 0.5 y perímetro 3.4
Sus puntos son: p1[x=0.0, y=0.0], p2[x=1.0, y=0.0], p3[x=0.0, y=1.0]
```

Modificar el constructor para que al crear una instancia de triángulo automáticamente se guarde en la propia instancia el tipo de triángulo que forma según sus lados.

Interfaces en JAVA

Interfaces en JAVA

- 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 solamente puede heredar como máximo de otra.
- Para poder ofrecer una funcionalidad muy similar a lo que sería la herencia múltiple JAVA dispone de interfaces.
- Podemos definir las **interfaces** como una colección de métodos abstractos y propiedades constantes, sin implementación. Se especifica **qué** se debe de hacer pero no **cómo**, serán las clases hijas quienes definan el comportamiento.
- Las interfaces tienen 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 de datos son static y final**. Se usa para definir valores constantes.

Interfaces en JAVA

- Para crear una Interfaz:

```
public interface NombreInterfaz {
  // constantes
  public static final tipo nombreConstante = valorConstante;

  // Métodos abstractos
  public tipo nombreMetodo(parámetros);
  public tipo nombreMetodo2(parámetros);
  // ...
}
```

- Para indicar que una clase implementa una interfaz:

```
public class nombreClase implements nombreInterfaz {
     //implementación métodos
     //...
}
```

Si una clase implementa una interfaz deberá sobreescribir todos sus métodos

Interfaces en JAVA

- Veamos un ejemplo:

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

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

```
public class Rectangulo implements Forma { ... }
```

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

Interfaces en JAVA

- Veamos un ejemplo:

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

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

     @override
     public double area()  { return ancho*largo; }
}
```

Interfaces en JAVA

- Veamos un ejemplo:

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

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

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

     @override
     public double area()  { return Math.PI*radio*radio; }

}
```

Múltiples Interfaces

- Una interfaz puede extender a otra interfaz, lo que significa que hereda todos los métodos y constantes definidos en la interfaz padre.

```
public interface NombreInterfazHija extends NombreInterfazPadre {
  // nuevos métodos y constantes

}
```

- Una clase puede implementar múltiples interfaces separándolas con comas en la declaración de la clase.

```
public class nombreClase implements nombreInterfaz1, nombreInterfaz2 {
    //implementación de los métodos de ambas interfaces
    //...
}
```

Múltiples Interfaces

- Veamos un ejemplo de implementación múltiple de 2 interfaces:

```
public class Cuadrado implements Forma, Color {

   private double lado;
   private String color;

   public Cuadrado(double lado, String color) {
       this.lado = lado;
       this.color = color;
   }
   public double area()      { return lado * lado; }

   public double perimetro() { return 4 * lado;    }

   public String color()     { return color;       }
}
```

```
public interface Forma {
    public double area();
    public double perimetro();
}

public interface Color {
    public String color();
}
```

Interfaces vs clases abstractas

- En las interfaces 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 interfaz sólo dispone de métodos abstractos.
- Como todos los métodos de una interfaz 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, podemos verlas como contratos, contratos donde está establecido qué debe hacer la clase que la implementa.

Tipos enumerados en JAVA

Tipos enumerados

- En JAVA, las clases tipo enum (enumeración) se utilizan para definir un conjunto fijo de valores. Los valores definidos por una clase enum se denominan "constantes de enumeración". Las clases tipo enum se utilizan comúnmente para definir tipos de datos que sólo pueden tener un conjunto específico de valores.

```
public enum DiaSemana { //posibles valores  }
```

- **Características**:
  - → Sobre un tipo enum tenemos disponible el método values() que devuelve un array con todos los valores del enum en el orden en que son declarados.
  - → 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 DiaSemana {
    LUNES,
    MARTES,
    MIÉRCOLES,
    JUEVES,
    VIERNES,
    SÁBADO,
    DOMINGO;
}
```

Tipos enumerados

- Las clases tipo enum también pueden tener métodos y constructores definidos. Por ejemplo, se puede agregar un método para obtener el número de días en un mes específico:

- La clase Mes define 12 constantes de enumeración que representan los meses del año. Además, la clase tiene un campo dias que se utiliza para almacenar el número de días en cada mes. La clase también tiene un constructor que se utiliza para inicializar el campo dias y un método getDias() que se utiliza para obtener el número de días en un mes específico.

```
public enum Mes {
    ENERO(31),
    FEBRERO(28),
    MARZO(31),
    ABRIL(30),
    MAYO(31),
    JUNIO(30),
    JULIO(31),
    AGOSTO(31),
    SEPTIEMBRE(30),
    OCTUBRE(31),
    NOVIEMBRE(30),
    DICIEMBRE(31);

    private final int dias;
    Mes(int dias)        { this.dias = dias; }
    public int getDias() { return dias;      }
}
```

Tipos enumerados

- En este ejemplo, se definen diaActual y mesActual, de tipo DiaSemana y Mes respectivamente. A continuación, se asignan los valores JUEVES y ENERO a partir de los nombres de las clases enumeradas.

```
public class AppTiposEnumerados1 {
    public static void main(String[] args) {
        String diasSemana = "";
        DiaSemana diaActual = DiaSemana.JUEVES;
        Mes mesActual = Mes.ENERO;
        for (DiaSemana d : DiaSemana.values()) diasSemana += d.toString() + ", ";
        System.out.println("Los días de la semana son: " + diasSemana);
        System.out.println("Hoy es "+ diaActual +" y el mes actual es " + mesActual);
        System.out.println(mesActual + " tiene " + mesActual.getDias() + " días");
    }
}
```

```
Los días de la semana son: LUNES, MARTES, MIÉRCOLES, JUEVES, VIERNES, SÁBADO, DOMINGO,
Hoy es JUEVES y el mes actual es ENERO
ENERO tiene 31 días
```

SALIDA POR PANTALLA

Tipos enumerados

- En ocasiones puede que debamos recibir información por consola y almacenarla como tipo enumerado. Para ello, deberemos guardar en una String un valor compatible con algún valor de tipo enumerado en cuestión y luego hacer una conversión.
- Haremos dicha conversión a través del método **valueOf(String s)** que contienen todas las clases de tipo **enum**, dicho método recibirá una String con un posible valor de tipo enumerado. Si el valor de la String no es válido, se producirá un error en ejecución.

```
public class AppTiposEnumerados2 {
    public static void main(String[] args) {
        Scanner entrada = new Scanner(System.in);
        System.out.println("Dime un mes " + Arrays.toString(Mes.values()));
        String s = entrada.nextLine();
        Mes mesActual = Mes.valueOf(s); // s → deberá ser un tipo enumerado válido
        System.out.println(mesActual + " tiene " + mesActual.getDias() + " días");
    }
}
```

```
//Ejemplo para entrada → ABRIL
ABRIL tiene 30 días
```

SALIDA POR PANTALLA

Comparar objetos : equals y hashCode

Comparar objetos: equals y hashCode

Los métodos equals() y hashCode() son importantes porque permiten que los objetos se comparen y se indexen correctamente en las colecciones de JAVA. La implementación correcta de estos métodos es esencial para garantizar que las colecciones funcionen correctamente y produzcan resultados esperados.

- **equals()**: se utiliza para comparar dos objetos y determinar si son iguales. Esta función es importante porque los objetos en JAVA se comparan por referencia, es decir, si dos objetos tienen la misma referencia (es decir, apuntan a la misma dirección de memoria), se consideran iguales. Sin embargo, en la mayoría de los casos, lo que se quiere es comparar el contenido de los objetos, en lugar de la referencia.

- **hashCode()**: se utiliza para obtener un valor numérico único que representa un objeto. Este valor se utiliza por el sistema de colecciones de JAVA para indexar y buscar objetos. Para implementar correctamente la función hashCode(), es necesario seguir las siguientes pautas:
    - Si la función equals() devuelve true para dos objetos, entonces los valores devueltos por la función hashCode() deben ser iguales.
    - Si dos objetos son diferentes según la función equals(), entonces sus valores devueltos por la función hashCode() pueden o no ser diferentes.

El método equals de la clase Object

- Para implementar correctamente la función equals(), es necesario sobrescribir el método de la clase base Object, dicha función se utiliza para determinar si dos objetos son iguales. Por lo que es importante que se implemente correctamente para que los objetos se comparen por su contenido y no por su referencia.
- Por otro lado, la implementación correcta de hashCode() debe garantizar que dos objetos que sean iguales según la función equals() devuelvan el mismo valor numérico, mientras que objetos diferentes pueden devolver diferentes valores.

```
public class Punto2D {
    private double x;
    private double y;
     ...
    public boolean equals(Object obj) {
        if (obj == null) return false;
        if (obj == this) return true;
        if (getClass() != obj.getClass()) return false;
        Punto2D otroPunto = (Punto2D) obj;
        return x == otroPunto.x && y == otroPunto.y;
    }
}
```

Los números de punto flotante (float y double) se representan internamente en binario. Como ya vimos en la segunda unidad, esto puede provocar en ocasiones pérdidas de precisión.
Por esta razón, se recomienda utilizar métodos como **Double.compare()** para comparar valores de punto flotante en lugar del operador **==**

El método equals de la clase Object

SIN SOBRESCRIBIR EL MÉTODO EQUALS

```
Punto2D p1 = new Punto2D(0, 0);
Punto2D p2 = new Punto2D(0, 0);
Punto2D p3 = new Punto2D(0, 0);
Punto2D p4 = p1;

if(p1.equals(p2)) System.out.println("Iguales");
else System.out.println("Distintos");
//Distintos

if(p1.equals(p4)) System.out.println("Iguales");
else System.out.println("Distintos");
//Iguales

ArrayList<Punto2D> puntos = new ArrayList<>();
puntos.add(p1);
puntos.add(p2);
puntos.add(p3);
puntos.add(p4);

puntos.remove(new Punto2D(0, 0));
//no borra ningún punto
```

MÉTODO EQUALS SOBRESCRITO

```
Punto2D p1 = new Punto2D(0, 0);
Punto2D p2 = new Punto2D(0, 0);
Punto2D p3 = new Punto2D(0, 0);
Punto2D p4 = p1;

if(p1.equals(p2)) System.out.println("Iguales");
else System.out.println("Distintos");
//Iguales

if(p1.equals(p4)) System.out.println("Iguales");
else System.out.println("Distintos");
//Iguales

ArrayList<Punto2D> puntos = new ArrayList<>();
puntos.add(p1);
puntos.add(p2);
puntos.add(p3);
puntos.add(p4);

puntos.remove(new Punto2D(0, 0));
//borra el primer punto con coordenadas (0,0)
```

El método hashCode de la clase Object

- El hashCode de un objeto en JAVA **es un valor entero que se utiliza para identificar el objeto de manera única en una colección de objetos**. El hashCode se utiliza como una clave en la implementación de estructuras de datos como HashMap, HashSet y otras colecciones de Java.
- El hashCode se debe basar en el contenido del objeto, es decir, en el estado de sus campos.
- La función hash está diseñada para mapear la entrada de manera eficiente a un valor hash único y consistente. Esto lo hace especialmente útil en la búsqueda de datos y la optimización de rendimiento. Se utilizan en colecciones basadas en hash, lo veremos en profundidad en el tema 7.

```
public class Punto {
    private int x; //1
    private int y; //2
     ...
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + x;
        result = prime * result + y;
        return result; //994
    }
}
```

```
public class Punto2D {
    private double x;
    private double y;
     ...
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        long temp;
        temp = Double.doubleToLongBits(x);
        result = prime * result + (int) (temp ^ (temp >>> 32));
        temp = Double.doubleToLongBits(y);
        result = prime * result + (int) (temp ^ (temp >>> 32));
        return result;
    }
}
```

Excepciones en JAVA

Excepciones

- Si en la ejecución de una parte del 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).

```
int dividendo = 10;
int divisor = 0;
int cociente = dividendo/divisor;
```

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

  - → 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, son las **excepciones**. Las excepciones en JAVA se dividen en dos categorías principales:
  - → **Excepciones verificadas** (checked exceptions), deben ser manejadas en tiempo de compilación. Son subclases que heredan de la clase **IOException**.
  - → **Excepciones no verificadas** (unchecked exceptions), no necesitan ser manejadas en tiempo de compilación. Son subclases que heredan de la clase **RuntimeException**.

Tipos de excepciones

Excepciones no verificadas (unchecked exceptions), no necesitan ser manejadas obligatoriamente en tiempo de compilación.

```
int num1 = 1 / 0; // Se producirá una ArithmeticException
int[] arr = new int[3];
int num2 = arr[3]; // Se producirá una ArrayIndexOutOfBoundsException
System.out.println(num1 + num2);
```

Excepciones verificadas (checked exceptions), deben ser manejadas obligatoriamente en tiempo de compilación.

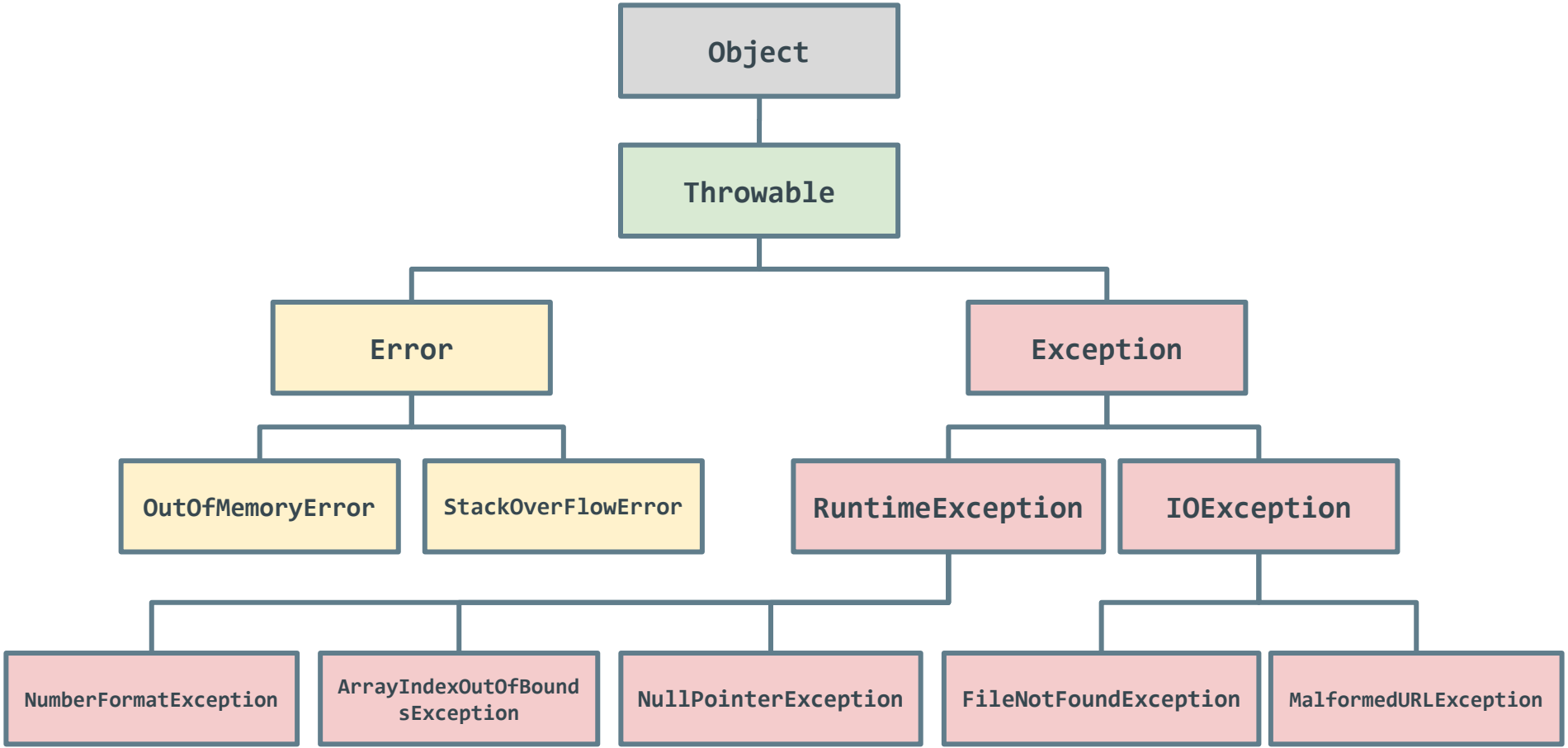
```
FileReader file = new FileReader("archivo.txt");
// El compilador mostrará un error, Unhandled exception type FileNotFoundException
```

```
try { FileReader file = new FileReader("archivo.txt"); }
catch (FileNotFoundException e) { System.out.println("ERROR"); }
// Error solucionado, estamos capturando la posible excepción
```

Excepciones en JAVA

- Una **excepción** en JAVA es un error o situación excepcional que se produce durante la ejecución de un programa. Java crea un objeto de excepción y lo lanza. A partir de ahí, el programa puede manejar la situación de diferentes maneras.
  - → No hacer nada.
  - → Capturar la excepción.
  - → Lanzarla y propagarla.
- Este mecanismo permite tratar los errores de una forma independiente, ya que **separa el código para el tratamiento de errores, del código normal del programa**.
- Todas las excepciones y errores en **JAVA** se representan, como vamos a ver en la siguiente sección, a través de **objetos** que **heredan**, en última instancia, de la clase **Throwable**.
  - → La clase **Throwable** define dos subclases principales: **Exception** y **Error**. La clase Exception representa excepciones que pueden ser manejadas y recuperadas, mientras que la clase Error representa problemas graves en la JVM o en el sistema en el que se está ejecutando.

La jerarquía Throwable



La jerarquía Throwable

Lista de algunas de las excepciones más frecuentes:

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

Capturar excepciones en JAVA

Capturar excepciones

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

- **try**: contiene las instrucciones donde se pueden producir las excepciones. Cuando una excepción se produce en el bloque try, en tiempo de ejecución JAVA busca un bloque catch que pueda manejar la excepción.
- **catch**: contiene las instrucciones a ejecutar para manejar una determinada excepción:
  - Se debe indicar el tipo de excepción se captura.
  - Si hubiese más de una excepción posible, se podrá incluir más de un bloque catch.
- **finally**: se ejecuta después del bloque try y del bloque catch, independientemente de si se produjo una excepción o no. El bloque finally se utiliza para liberar recursos, cerrar archivos y realizar otras tareas importantes.

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 e) {
    //bloque3
} catch (NullPointerException e) {
    //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**

Ejemplos de captura de excepciones

```
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");
     }
}
```

```
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");
     }
     System.out.println("El programa imprime esta frase con normalidad");
}
```

Capturar múltiples excepciones

catch

- Podemos especificar todos los tipos de excepciones queramos gestionar.
- O bien, podemos capturar cualquier excepción indicando la clase **Exception**, ya que es la superclase común de todas las excepciones.
- Métodos útiles 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.

```
try {
     ...
     //Aquí va el código que puede lanzar una excepción
} catch (TipoException1 e1) {
     System.out.println("El error es: " + e1.getMessage());
} catch (TipoException2 e2) {
     System.out.println("El error es: " + e2.getMessage());
} catch (TipoException3 e3) {
     System.out.println("El error es: " + e3.getMessage());
}
```

Ejemplos de captura de excepciones

```
public static void main (String args[]) {
     Scanner entrada = new Scanner(System.in);
     int num1, num2;
     try {
          num1 = entrada.nextInt();
          num2 = Integer.parseInt("hola");
          System.out.println(num2 / num1);
     } catch (ArithmeticException e) {
          System.out.println("Error: " + e.getMessage());
          e.printStackTrace();
     }
     catch (InputMismatchException e) {
          System.out.println("Error: " + e.getMessage());
          e.printStackTrace();
     }
     catch (NumberFormatException e) {
          System.out.println("Error: " + e.getMessage());
          e.printStackTrace();
     }
     System.out.println("El programa termina con normalidad");
}
```

Excepciones con bloque finally

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.

```
try {
     int a = 1 / 0;
     System.out.println(a);

} catch (ArithmeticException e) {
     System.out.println("Error");
}
finally {
     System.out.println("Fin del programa");
}
System.out.println("Esto se muestra");
```

```
try {
     int a = 1 / 0;
     System.out.println(a);

} catch (InputMismatchException e) {
     System.out.println("Error");
}
finally {
     System.out.println("Fin del programa");
}
System.out.println("Esto NO se muestra");
```

Lanzar y propagar excepciones en JAVA

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");
     }

     System.out.println("Los minutos introducidos son válidos");
}
```

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.

```
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**).

Método que propaga la excepción

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

Un método que NO propaga la excepción

```
public void leer() {
     try {
          //Fragmento de código que puede
          //lanzar una excepción de tipo IOException
     } catch (IOException e) {
          //Tratamiento de la excepción
     }
}
```

Lanzar y propagar una excepción

Ejemplo:

```
static void dividir(int n, int m) throws ArithmeticException {

     if (m == 0)
          throw new ArithmeticException(); //lanzamos la excepción
     else
          System.out.println(n + "/" + m + "=" + (n / m));
}

public static void main(String args[]) {
     try {
          dividir(3, 0);
     } catch (ArithmeticException e) {
          System.out.println("Capturando excepción en el main");
     }
}
```

Lanzar y propagar una excepción

Ejemplo:

```
static void dividir(int n, int m) throws ArithmeticException {
     try {
          if (m == 0)
               throw new ArithmeticException(); //lanzamos la excepción
          else
               System.out.println(n + "/" + m + "=" + (n / m));
     } catch (ArithmeticException e) {
          System.out.println("Capturando excepción dentro de un método");
     }
}

public static void main(String args[]) {
     try {
          dividir(3, 0);
     } catch (ArithmeticException e) {
          System.out.println("Capturando excepción en el main");
     }
}
```

Lanzar y 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: 0-2");
            pos = entrada.nextInt();
            try {
                  ejemplo(pos, elArray);
                  salir = true;
            } catch (ArrayIndexOutOfBoundsException e) {
                  System.out.println("Se ha producido una excepción: " + e );
            }
      }while(!salir);
}
```

Propagar excepción múltiple en JAVA

Propagar más de una excepción

```
static Scanner entrada = new Scanner(System.in);
public static int obtenerDatos(int[] v) throws ArrayIndexOutOfBoundsException, InputMismatchException {
    System.out.println("Dime una posición entre 0 y "+(v.length-1));
    int n = entrada.nextInt();  //posible excepción de tipo InputMismatchException
    if (n < 0 || n >= v.length) //posible excepción de tipo ArrayIndexOutOfBoundsException
        throw new ArrayIndexOutOfBoundsException("ERROR: índice no válido");
    return n;
}
public static void main(String args[]) {
    boolean salir = false;
    int pos;
    int[] array = { 15, 2, 8, 19, 8 };
    do {
        try {
            pos = obtenerDatos(array);
            System.out.println("array[" + pos + "] = " + array[pos]);
            salir = true;
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println(e.getMessage());
        } catch (InputMismatchException e) {
            System.out.println("ERROR: número no válido");
            entrada.next();
        }
    } while (!salir);
    entrada.close();
}
```

Al ser excepciones no verificadas, no es necesario manejarlas.
Podemos prescindir tanto del **try - catch** como del **throws**. Sin embargo, es una buena práctica manejar adecuadamente las excepciones.

Excepciones de usuario en JAVA

Excepciones de usuario

En JAVA, existe la posibilidad de definir **excepciones de usuario** para representar errores de lógica de la aplicación, para ello:

1. Se debe **crear una clase que herede de Exception o RuntimeException**
2. Añadir los **métodos** y **propiedades** necesarios para almacenar información relativa a nuestro tipo de error. Entre ellos los posibles constructores utilizados al lanzar la excepción(**throw new …**)
3. Si la excepción de usuario extiende de la clase **Exception**, entonces es una excepción **verificada**, lo que significa que el compilador verifica que se maneje adecuadamente en el código. En este caso, el método que lanza la excepción debe declarar la cabecera del método con la cláusula "throws" o capturar la excepción usando un bloque "try-catch".
4. Si la excepción de usuario extiende de la clase **RuntimeException**, entonces es una excepción **no verificada**, lo que significa que el compilador no requiere que se maneje en el código. En este caso, no es necesario declarar la excepción en la cabecera del método ni capturarla en un bloque "try-catch".
5. En general, se recomienda utilizar excepciones verificadas.

Excepciones de usuario

```
public class ClienteExisteException extends Exception
{
     private Cliente cliente;
     //constructor utilizado en throw
     public ClienteExisteException (Cliente cliente) {
          this.cliente = cliente;
     }
     public String toString() {
          return "El cliente de nombre " + this.cliente.getNombre() + " ya existe";
     }
}
```

```
public class ClienteExisteException extends RuntimeException
{
     private Cliente cliente;
     //constructor utilizado en throw
     public ClienteExisteException (Cliente cliente) {
          this.cliente = cliente;
     }
     public String toString() {
          return "El cliente de nombre " + this.cliente.getNombre() + " ya existe";
     }
}
```

Excepciones de usuario

Ejemplo: comprobamos si un cliente existe a partir de su nombre, se utiliza la excepción verificada:

```
public static void nuevoCliente(String nombre, ArrayList<Cliente> clientes) throws ClienteExisteException
{
      for(Cliente c : clientes) {
            if(c.getNombre().equals(nombre))
               throw new ClienteExisteException(c);
      }
      clientes.add(new Cliente(nombre));
      System.out.println("Cliente dado de alta correctamente");
}
```

Al invocar al método nuevoCliente(...) debemos capturar la excepción correspondiente al **throws** indicado en la cabecera del método. En este ejemplo **ClienteExisteException.**

Otra opción sería incluir el bloque **try - catch** dentro del propio método nuevoCliente(...)

```
ArrayList<Cliente> clientes = new ArrayList<>();
clientes.add(new Cliente("Pep"));
clientes.add(new Cliente("Tom"));
clientes.add(new Cliente("Kal"));
try {
      nuevoCliente("Tom", clientes);
      nuevoCliente("Eddie", clientes);
} catch (ClienteExisteException e) {
      System.out.println("Error: " + e);
}  //no se añade ningún cliente
```

Excepciones de usuario

```java
public class PuntoNoValidoException extends Exception {
      private Punto punto;
      public PuntoNoValidoException (Punto punto) {  this.punto = punto;     }
      public String toString() { return "Punto(" + punto.getX() + ", " + punto.getY() + ") no válido";  }
}
```

MAIN

```java
static Scanner entrada = new Scanner(System.in);
public static Punto nuevoPunto() throws PuntoNoValidoException, InputMismatchException
{
      System.out.println("Dime x");
      int x = entrada.nextInt();       //posible excepción de tipo InputMismatchException
      System.out.println("Dime y");
      int y = entrada.nextInt();       //posible excepción de tipo InputMismatchException
      Punto p = new Punto(x,y);
      if (x < 0 || y < 0)              //posible excepción de tipo PuntoNoValidoException
            throw new PuntoNoValidoException (p);
      System.out.println("Punto(" + x + "," + y + ") creado correctamente");
      return p;
}
public static void main(String[] args) {
      try   { nuevoPunto(); }
      catch (PuntoNoValidoException e) { System.out.println("Error: " + e); }
      catch (InputMismatchException e) { System.out.println("Error: Valor numérico no válido"); }
}
```

Excepciones de usuario

```
public class PuntoNoValidoException extends Exception {
      private Punto punto;
      public PuntoNoValidoException (Punto punto) {  this.punto = punto;      }
      public String toString() { return "Punto(" + punto.getX() + ", " + punto.getY() + ") no válido";  }
}
```

```
public class Punto {
      private int x;
      private int y;
      public Punto (int x, int y) throws PuntoNoValidoException
      {
            this.x = x;
            this.y = y;
            if (x < 0 || y < 0) throw new PuntoNoValidoException(this);
      }
      //getters, setters y métodos
      //...
}
```

MAIN

```
public static void main(String[] args) {
      try { Punto p = new Punto(1,-1); }
      catch (PuntoNoValidoException e) { System.out.println("Error: " + e); }
}
//Salida del main → Error: Punto(1, -1) no válido
```

Ejercicios Herencia y excepciones en JAVA: AppSucursalBancaria

Ejercicios Herencia y excepciones en JAVA

1. Ampliar la aplicación **AppSucursalBancaria** desarrollada en la unidad anterior. Se almacenarán datos de clientes y los datos relativos a sus cuentas a través de las clases **Cuenta** y **Cliente**.
2. Todas las cantidades se deben manejar con el tipo de datos “double”.
3. Añadir tres tipos de cuenta: CuentaCorriente (CC), CuentaVivienda (CV) y FondoInversion (FI).

   CC: Se crea con 0€.

   CV: Se crea con 1000€.

   FI: Se crea con 5000€. Se deberá almacenar el interés anual generado.

4. Al crear la cuenta, se deberá preguntar el tipo de cuenta que se desea abrir, ofreciendo los 3 tipos. Se deberá preguntar de la forma que se considere menos propensa a errores y más fácil de responder.
5. Al mostrar los datos de la cuenta se deberá mostrar el tipo de cuenta asociada. Si se trata de un FI se deberá mostrar el interés anual generado por dicha cuenta.

Ejercicios Herencia y excepciones en JAVA

6. Implementa la gestión de excepciones, podrás crear todas las excepciones de usuario que consideres necesarias. Debes controlar que se lance una excepción en los siguientes casos:
    a. Se intenta insertar un cliente que ya existe en la sucursal.
    b. Fecha de nacimiento del cliente no válida.
    c. Cantidad del retiro no válida para el tipo de cuenta.
        i. **CC**: Retiro máximo 300€, no se puede retirar si el saldo final es menor de 0€.
        ii. **CV**: Retiro máximo 500€, no se puede retirar si el saldo final es menor de 0€.
        iii. **FI**: Retiro mínimo 500€, no se puede retirar si el saldo final es menor de 3000€.
    d. Cantidad del ingreso no válida para el tipo de cuenta.
        i. **CC**: Ingreso mínimo de 10€
        ii. **CV**: Ingreso mínimo de 10€
        iii. **FI**: Ingreso mínimo de 500€
    e. Opción no válida en algún menú (por ejemplo, se inserta un carácter en vez de un número).
    f. Otras comprobaciones adicionales que consideres necesarias.

Bibliografía:

Allen Weiss, M. (2007). *Estructuras de datos en JAVA*. Madrid: Pearson

Froufe Quintas, A. (2002). *JAVA 2: Manual de usuario y tutorial*. Madrid: RA-MA

J. Barnes, D. *Programación orientada a objetos en JAVA*. Madrid: Pearson

Desing Patterns. Elements of Reusable. OO Software

JAVA Limpio. Pello Altadi. Eugenia Pérez

Apuntes de Programación de Anna Sanchis Perales

Apuntes de Programación de Lionel Tarazón Alcocer

Ilustraciones:

https://pixabay.com/

https://freepik.es/

https://lottiefiles.com/

Preguntas