

# 1º CFGS DESARROLLO DE APLICACIONES MULTIPLATAFORMA

## UD 5. Programación orientada a objetos. Introducción.

Módulo: Programación



Centro de Enseñanza  
Gregorio Fernández

# Métodos

- Un método es un conjunto de instrucciones que realizan una tarea concreta y que se agrupan bajo un identificador.
- Un método puede ser visto como una caja negra, con unas entradas sobre las que realiza unas operaciones y una salida que genera como resultado.
- Para ejecutar las instrucciones de un método hay que **llamar** al método, escribiendo su identificador en el código fuente del programa.
- La utilidad principal de los métodos es la **reutilización de código**, ya que un método puede ser utilizado muchas veces desde un mismo programa o desde otros programas sin tener que volver a escribir su código.
- Otra utilidad es **mejorar la legibilidad del código**.

# Métodos

- Sintaxis de un método:

```
[visibilidad] [static] tipo_retorno nombre ([parámetros])  
{  
    //cuerpo del método  
}
```

- La **visibilidad** es alguno de los modificadores de acceso vistos.
- El método puede ser o no **estático**.
- El **tipo de retorno** se refiere al tipo de datos devuelto por el método, o **void** si no devuelve nada.
- La lista de **parámetros** se indica de la siguiente forma:

**tipo1 param1, tipo2 parm2, ...**

# Métodos

## I – Sentencia return

- Sirve para devolver un valor desde el método.
- El valor puede ser un literal, una variable o el resultado de una expresión.
- El valor returned debe ser del mismo tipo que el especificado en la definición del método.
- Se pueden escribir varias sentencias **return** en el interior del método y en distintas posiciones, aunque hay que tener cuidado de no dejar algún fragmento de código inalcanzable.
- Ejemplo:

```
public int suma(int a, int b) {  
    int resultado = a + b;  
    return resultado;  
}
```



# Métodos

## II – Llamada

- Para hacer uso del método hay que **invocarlo (llamarlo)**.
- Esto se hace escribiendo en el código fuente el nombre del método seguido de (entre paréntesis) tantos parámetros y del mismo tipo como los que se especifiquen en la definición del método.
- Si no hay lista de parámetros se pondrán los paréntesis vacíos.
- En el momento de la llamada al método, el flujo de ejecución del programa pasa de la sentencia de llamada al interior del método, y éste retiene el flujo hasta que se produce el retorno mediante una instrucción return o hasta que termina el cuerpo del método.
- Posteriormente, el flujo de ejecución retorna de nuevo a la línea siguiente desde la cual se llamó al método, continuándose la ejecución del programa.



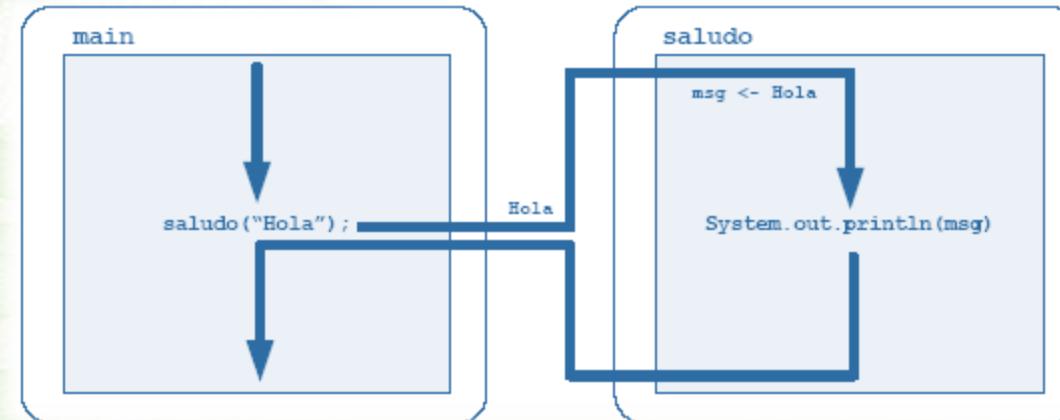
# Métodos

## II – Llamada

- Veamos gráficamente el proceso de llamada al método **saludo**:

```
public static void main(String[] args) {  
    saludo(Hola);  
}
```

```
public static void saludo(String msg) {  
    System.out.println(msg);  
}
```

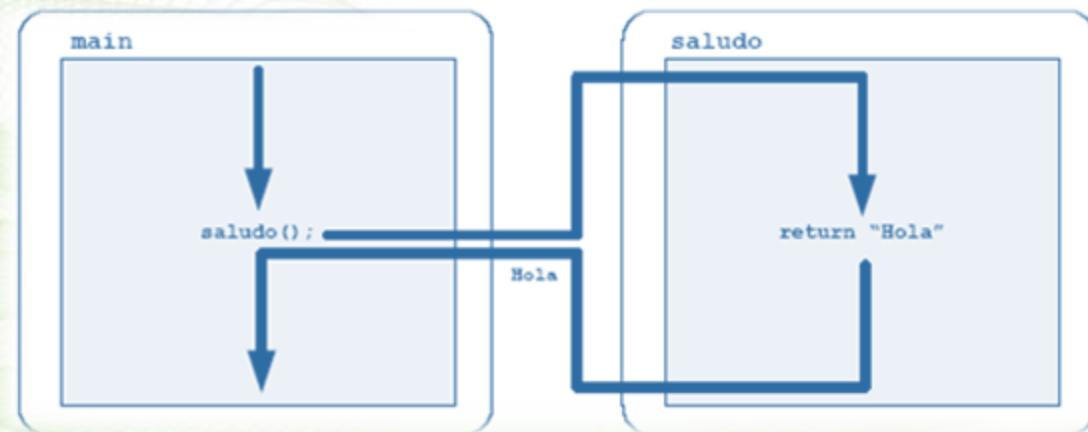


# Métodos

## II – Llamada

- Modificamos el método **saludo** de la siguiente forma:

```
public static void main(String[] args) {  
    String msg = saludo();   
    System.out.println(msg);  
}  
  
public static String saludo() {  
    return("Hola");  
}
```



# Métodos

## III – Paso de parámetros

- **Parámetros formales:** los definidos en la declaración de un método.
- **Parámetros actuales:** los pasados en la llamada al método.
- Cuando se invoca a un método, los parámetros actuales **se copian** en los parámetros formales.
- Estos dos tipos de parámetros se encuentran en ámbitos diferentes.
- Analicemos el siguiente código:



# Métodos

## III – Paso de parámetros

```
public double raizCuadrada(double n) {  
    if(n>0){  
        return Math.sqrt(n);  
    }  
    System.out.println("Error. Número negativo");  
    return -1;  
}  
  
public static void main(String[] args) {  
    Pruebas app=new Pruebas();  
    double num=10;  
    double raiz=app.raizCuadrada(num);  
    n=123;  
}
```



# Métodos

## III – Paso de parámetros

- Hacemos un cambio:

```
public double raizCuadrada(double n) {  
    n=--num;  
    if(n>0) {  
        return Math.sqrt(n);  
    }  
    System.out.println("Error. Número negativo");  
    return -1;  
}  
  
public static void main(String[] args) {  
    Pruebas app=new Pruebas();  
    double num=10;  
    double raiz=app.raizCuadrada(num);  
}
```



# Métodos

## III – Paso de parámetros

- Y con este cambio, cuanto vale la variable **raiz**?

```
double num=10;  
public double raizCuadrada(double n) {  
    n=--num;  
    if(n>0) {  
        return Math.sqrt(n);  
    }  
    System.out.println("Error. Número negativo");  
    return -1;  
}  
  
public static void main(String[] args) {  
    Pruebas app=new Pruebas();  
    double raiz=app.raizCuadrada(num);  
}
```



# Métodos

## III – Paso de parámetros

- Java admite dos formas de pasar parámetros en las llamadas a los métodos dependiendo del **tipo de los parámetros**:
- Paso de parámetros **por valor** → **tipos primitivos**
  - Se pasa una **COPIA DEL VALOR** de los parámetros actuales a los parámetros formales.
  - La información estará duplicada en memoria.
  - Las modificaciones realizadas en el método no afectan a los parámetros actuales.
- Paso de parámetros **por referencia** → **tipos objeto**
  - Se pasa una **COPIA DE LA DIRECCIÓN DE MEMORIA** del objeto (parámetro actual).
  - Parámetro actual y parámetro formal “apuntan” a la misma dirección de memoria en la cual se está almacenado el objeto.
  - Las modificaciones afectan al parámetro actual.



# Métodos

## III – Paso de parámetros

- Analicemos el siguiente ejemplo de **PASO POR VALOR**:

```
public void metodo(int i) {  
    System.out.println("En el método - Antes i=" + i);  
    i++;  
    System.out.println("En el método - Después i=" + i)  
}  
  
public static void main(String[] args) {  
    Pruebas app=new Pruebas();  
    int n=0;  
    System.out.println("En el main - Antes n=" + n);  
    app.metodo(n);  
    System.out.println("En el main - Después n=" + n);  
}
```



# Métodos

## III – Paso de parámetros

- Veamos ahora un ejemplo de **PASO POR REFERENCIA**:

```
public void metodo(MiClase obj) {  
    System.out.println("En el método - Antes a=" + obj.a);  
    obj.a++;  
    System.out.println("En el método - Despues a=" + obj.a)  
}  
  
public static void main(String[] args) {  
    Pruebas app=new Pruebas();  
    MiClase mC=new MiClase();  
    mC.a=0;  
    System.out.println("En el main - Antes a=" + mC.a);  
    app.metodo(mC);  
    System.out.println("En el main - Despues a=" + mC.a);  
}
```

```
public class MiClase {  
    public int a;  
}
```

# Métodos

## IV – Recursividad

- Técnica de programación consistente en que un método **se llame a sí mismo**.
- Permite la realización de algoritmos elegantes.
- Veamos un primer ejemplo:

```
public void contar(int i){  
    System.out.print(i + " ");  
    contar(++i); ← Llamada recursiva  
}
```

- Este método es **recursivo** porque se llama a sí mismo.
- Recibe un número entero, lo imprime por pantalla y se vuelve a llamar a sí mismo, pero pasando el valor recibido como parámetro incrementado en 1.
- Para ejecutarlo hay que llamarlo una **primera vez** pasando un valor inicial:

```
contar(0);
```

- Que ocurre al ejecutar?

# Métodos

## IV – Recursividad

- Para que un método recursivo no deje un programa “colgado” por **recursividad infinita**, es necesario que exista una **condición de finalización**.
- Esta condición (**if**), hará que la secuencia de llamadas recursivas llegue a su fin y que cada método recursivo finalice de manera natural.
- En el ejemplo anterior la condición de finalización podría ser que cuando imprima el número 99 finalice:

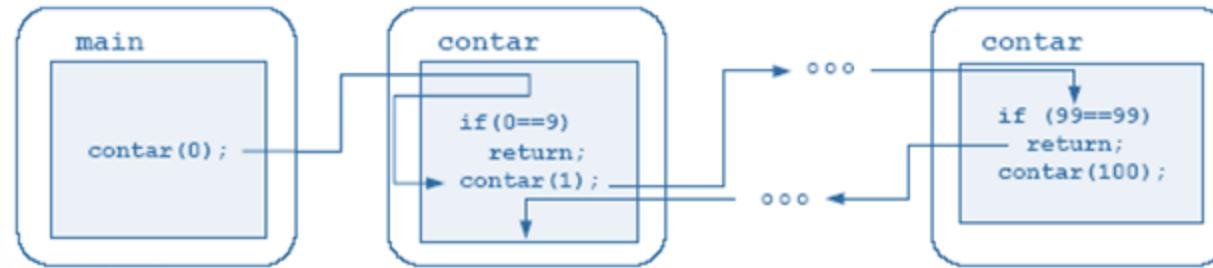
```
public void contar(int i) {  
    System.out.print(i + " ");  
    if(i==99) return;  
    contar(++i);  
}
```



# Métodos

## IV – Recursividad

- Veamos la secuencia de llamadas recursivas, de forma gráfica:



- En cada una de las llamadas recursivas los métodos quedan “**latentes**” a la espera de que finalice el método al que han llamado, que aunque sea el mismo, es como si fuera un método totalmente diferente.
- Cuando se termina esa espera porque el método invocado finaliza, cada uno de los métodos invocadores continúa su ejecución, de forma que como ya no existen más sentencias terminan de manera natural, retornando el flujo de ejecución al método superior que lo invocó.
- Este proceso de **vuelta a atrás** continúa hasta que el flujo llega al método main y el programa finaliza.

# Métodos

## IV – Recursividad

- Un ejemplo clásico de recursividad en programación es el algoritmo del **factorial** de un número.
- El factorial de un número N (representado por  $N!$ ), es el producto de todos los números naturales menores o iguales a N:

$$N! = N * (N-1) * (N-2) * \dots * 3 * 2 * 1$$

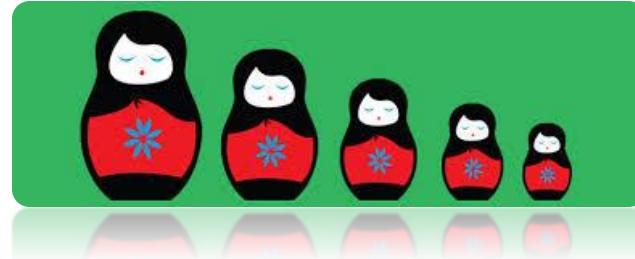
- Por ejemplo  $4! = 4 * 3 * 2 * 1 = 24$ .
- El factorial de 0 se define como 1 ( $0! = 1$ ).
- La implementación en java podría ser:

```
public int factorial(int n) {  
    if(n==0) return 1;  
    return n * factorial(n-1);  
}
```



# Métodos

## IV – Recursividad



- Cuestiones a tener en cuenta:

- ✓ Cada llamada recursiva implica que se reserva memoria para las variables del “nuevo” método. Esto implica que si se cae en una recursividad infinita, el programa provocará un error de ejecución por **desbordamiento de la pila** de memoria.
- ✓ Las llamadas recursivas deben hacerse de forma que en cada llamada se resuelva un **problema más sencillo**.  
Es más sencillo resolver factorial(4) que factorial(5).
- ✓ Todo problema recursivo debe tener una **condición de finalización** y ésta debe ser **alcanzable** en un número finito de pasos.
- ✓ Todo problema recursivo puede resolverse **de forma no recursiva** (iterativa).
- ✓ En ocasiones usando recursividad, es **más sencillo implementar** soluciones.
- ✓ No olvidar, que es una técnica que **consume memoria**.

# POO - Introducción

- Dos de los paradigmas de programación más difundidos son:
  - Modelo **procedimental** (programación imperativa u orientado a procesos)
  - Modelo **orientado a objetos**.
- El **modelo procedimental** está organizado en torno a los procesos.
- El **modelo orientado a objetos** está orientado a los datos (objetos), los cuales son considerados más importantes que los procesos.
- La potencia de la POO (programación orientada a objetos) está en que hace más sencillo resolver problemas, al dividir el problema en objetos, de forma que cada objeto funciona de forma totalmente independiente.



# Objetos

- Java es un lenguaje totalmente orientado a objetos, de forma que el objeto es la entidad fundamental del lenguaje.
- Los objetos son creados a partir de una **clase**.
- Una clase, se puede ver como el tipo de datos del objeto.
- Características de un objeto:
  - **Identidad.** Cada objeto es único y diferente.
  - **Estado.** El estado de un objeto son los valores de sus atributos en un momento determinado.
  - **Comportamiento.** Son los métodos del objeto.



# Objetos

- El ciclo de vida de un objeto consta de las siguientes fases:
  1. Creación.
  2. Uso.
  3. Destrucción.



# Objetos

## I - Creación

- Declarar + Instanciar.
- La **declaración** de un objeto consiste en la declaración de una variable del tipo de la clase del objeto → **referencia**
- Después de declarar el objeto, hay que instanciarlo para poder usarlo.
- La **instanciación** consiste en la creación en sí del objeto, es decir, ponerlo en memoria.
- Para instanciar un objeto se necesita llamar a un **constructor** de la clase mediante el operador **new**.
- Si intentamos usar un objeto sin antes haberlo instanciado se lanza la excepción **NullPointerException**.



# Objetos

## II - Uso

- Una vez creado el objeto podremos usarlo:
  - Accediendo a sus variables (de instancia).
  - Invocando a sus métodos.
- Para referirse a los miembros de un objeto se utiliza el operador punto “.”:

**referencia.miembro**



# Objetos

## III - Destrucción

- Los destructores de objetos se utilizan generalmente, para liberar recursos y cerrar flujos abiertos.
- No reciben parámetros.
- No está permitida su sobrecarga.
- En Java no hay destructores como en otros lenguajes de programación.



# Beneficios de la POO

- **Modularidad.** El código fuente de un objeto puede mantenerse y reescribirse sin que haya que reprogramar el código de otros objetos de la aplicación.
- **Reutilización de código.** Podemos utilizar clases y objetos de terceras personas. No tenemos que conocer los detalles de su implementación interna sino solamente su interfaz.
- **Facilidad de testeo y reprogramación.** Si tenemos un objeto que está dando problemas en una aplicación, no tenemos que reescribir el código de toda la aplicación, sino que tenemos que reemplazar el objeto problemático por otro similar, o bien reprogramarlo.
- **Ocultación de la información.** Se ocultan los detalles de implementación. Lo que interesa es la interfaz.



# Propiedades de la POO

- **Abstracción.** Según la RAE, abstraer es:

*“separar por medio de una operación intelectual las cualidades de un objeto para considerarlas aisladamente o para considerar el mismo objeto en su pura esencia o noción”.*

Cuando se programa orientado a objetos lo que se hace es abstraer las características de los objetos que van a formar parte del programa, y crear las clases con sus atributos y sus métodos.

- **Encapsulamiento.** Los objetos se ven según su comportamiento externo. Podemos ejecutar los métodos de un objeto sin tener por qué saber cómo funciona internamente el método.

# Propiedades de la POO

- **Ocultación de la información.** Un objeto está aislado del exterior, de forma que su zona privada sólo es utilizada por métodos de la propia clase, y expone su zona pública (llamada interfaz de la clase) para que pueda ser utilizada por otros objetos.
- **Herencia.** Las clases se estructuran formando jerarquías, de tal forma, que una clase (subclase) puede heredar propiedades de otra clase (superclase).
- **Polimorfismo.** El polimorfismo permite crear métodos compatibles con objetos de distintos tipos. Es un concepto íntimamente relacionado con la herencia.

# Clases

- Una clase es la **definición** de un tipo de objeto.
- **Prototipo o plantilla** que define las propiedades y métodos comunes a todos los objetos de esa clase, a partir de la cual se crean objetos.
- A partir de una clase podremos crear objetos de esa clase.
- Una clase es un concepto **abstracto** que no ocupa memoria. Un objeto, sin embargo, es un concepto **real**. Cada objeto tiene su propio espacio en memoria para almacenar sus datos y sus métodos.



# Clases

## I - Creación

- **Definir:**
  - **Atributos (campos ó datos).** Es decir, los datos miembros de esa clase. Pueden ser públicos (accesibles desde otra clase) o privados (sólo accesibles por código de su propia clase).
  - **Métodos.** Acciones (operaciones) que puede realizar la clase.
  - **Código de inicialización.** Operaciones previas a la construcción de la clase (método constructor).



# Clases

## I - Creación

- Sintaxis:

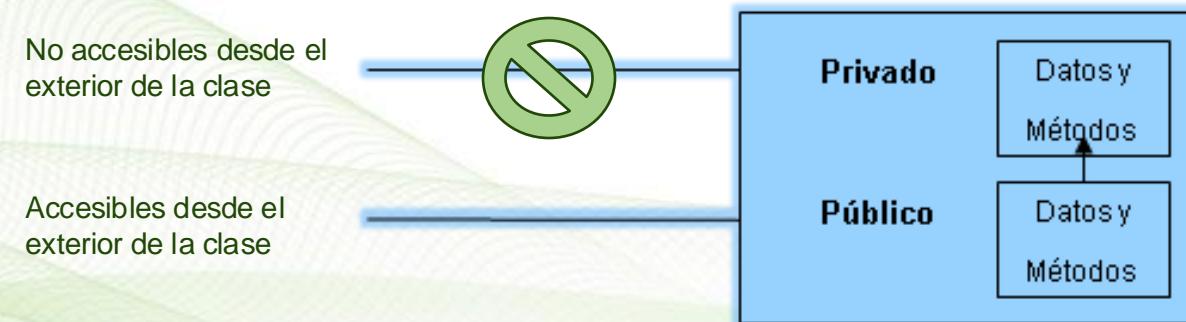
```
[acceso] class nombreClase {  
    [acceso] [static] tipo atributo1;  
    [acceso] [static] tipo atributo2;  
    [acceso] [static] tipo atributo3;  
    ...  
    [acceso] [static] tipo método1([listaDeArgumentos]) {  
        .....  
    }  
    ...  
}
```



# Clases

## II - Encapsulamiento

- Uno de los principios fundamentales de la POO es la **ocultación de la información**, lo cual significa que no es posible acceder a miembros de la clase desde el exterior.
- Esto se consigue haciendo esos miembros privados.



# Clases

## II - Encapsulamiento

- Para controlar el acceso a los miembros de una clase, se utilizan los modificadores de acceso **public**, **private** y **protected**. Éstos, determinan la visibilidad de los miembros de la clase:

Acceso	Miembro de la misma clase	Miembro de una clase derivada	Miembro de la clase del paquete	Miembro de la clase de otro paquete
<b>private</b>	✓	✗	✗	✗
<b>ninguno</b>	✓	✗	✓	✗
<b>protected</b>	✓	✓	✓	✗
<b>public</b>	✓	✓	✓	✓

- A los miembros **public** se puede acceder por cualquier método desde fuera de la clase.
- A los miembros **private** sólo se puede acceder por métodos de la misma clase.
- Y a los miembros **protected**, se puede acceder por métodos de la misma clase ó de clases derivadas, así como por métodos de otras clases que se encuentran en el mismo paquete (se verá con más detalle cuando hablemos de la “herencia”).



# Clases

## II - Encapsulamiento

- Principio de **ocultación de la información**:

*Toda interacción con un objeto se debe restringir a utilizar una **interfaz bien definida**, que permita que los detalles de implementación de los objetos sean ignorados.*

*Por consiguiente, los **elementos públicos** forman la interfaz externa del objeto, mientras que los **elementos privados** son aspectos internos del objeto que no necesitan ser accesibles para usar el objeto.*



# Refencia “this”

- Referencia al objeto con el que se está trabajando.
- Es decir, se refiere al objeto que está ejecutando el método.
- En algunas ocasiones sirve para resolver ambigüedades o para devolver referencias al propio objeto.
- Lo vemos en el siguiente ejemplo:



# Refencia “this”

```
public class Rectangulo {  
  
    private int ancho;  
    private int alto;  
  
    public Rectangulo(int ancho, int alto){  
        this.ancho=ancho;  
        this.alto=alto;  
    }  
  
    public int getAncho(){  
        return this.ancho;  
    }  
  
    public int getAlto(){  
        return this.alto;  
    }  
  
    public Rectangulo incrementarAncho(){  
        this.ancho++;  
        return this;  
    }  
  
    public Rectangulo incrementarAlto(){  
        this.alto++;  
        return this;  
    }  
}
```



# La clase Object

- Raíz jerárquica de Java.
- Cualquier clase es subclase de **Object**.
- Dispone de una serie de métodos que pueden ser utilizados por todos los objetos, ya que los heredan.
- Consultar el API de java.
- Nosotros vamos a con más profundidad los métodos siguientes:
  - **toString()**
  - **equals()**
  - **getClass()**



# La clase Object

## I – Método `toString()`

- Devuelve la representación **textual** del objeto.
- Normalmente las clases redefinen este método para obtener el resultado esperado.
- Ejemplo:

```
Rectangulo r1=new Rectangulo(5,7);  
System.out.println(r1.toString());
```



¿Es lo que  
esperaba?



# La clase Object

## II – Comparar objetos

- Podemos usar el método **equals()**.
- Ejemplo:

```
Rectangulo r1=new Rectangulo(5,3);  
Rectangulo r2=new Rectangulo(5,3);  
System.out.println(r1.equals(r2));
```



- Y ahora?:

```
r1=r2;  
System.out.println(r1.equals(r2));
```

# La clase Object

## II – Comparar objetos

- Si queremos hacer una comparación de objetos “en profundidad”, tenemos que sobrescribir el método **equals** y determinar el criterio de comparación.
- En nuestro ejemplo:

```
public boolean equals(Rectangulo r) {  
    return r.getAlto() == this.alto &&  
        r.getAncho() == this.ancho;  
}
```



# La clase Object

## II – Comparar objetos

- Si con el método `equals` no tenemos suficiente, podemos comparar objetos con el método `compareTo()`.
- Si por ejemplo quisiéramos ordenar “Rectángulos”.
- Para ello, la clase debe implementar la interfaz **Comparable**:

```
public class Rectangulo implements Comparable<Rectangulo>
```

- Y posteriormente, implementar el método `compareTo`:

```
public int compareTo(Rectangulo o) {  
    if (this.equals(o)) {  
        return 0;  
    }  
    if (this.ancho > o.getAncho() && this.alto > o.getAlto()) {  
        return 1;  
    }  
    return -1;  
}
```



# La clase Object

## III – Método getClass()

- Devuelve la **clase del objeto** en tiempo de ejecución .
- Concretamente, devuelve un objeto **Class** al que se le puede pedir información sobre la clase: nombre, quién es su superclase,...

```
public void mostrarInf(Object obj) {  
    System.out.println("Nombre de la clase: " + obj.getClass().getName());  
    System.out.println("Superclase: " + obj.getClass().getSuperclass());  
}
```

- Un uso práctico del método getClass(), podría ser crear un ejemplar de una clase sin conocer la clase en tiempo de compilación.

```
public Object crearInstanciaDe(Object obj) {  
    return obj.getClass().newInstance();  
}
```



# Constructores



- Un **constructor** es un método no estático que se ejecuta automáticamente cuando se crea un objeto de la clase. Sirve para **inicializar** los datos de la clase.
- Reglas:
  - Método no estático.
  - Tiene el mismo nombre que la clase.
  - Puede tener cero o más argumentos.
  - No devuelve ningún valor (ni siquiera void).
  - Debe ser público.

# Constructores

## I – Por defecto

- Constructor que **no tiene parámetros**.
- Normalmente da valor inicial a los datos de la clase asignándoles valores por defecto.
- Ejemplo:

```
public class Punto {  
    private double x;  
    private double y;  
  
    public Punto() {  
        x=0.0;  
        y=0.0;  
    }  
}
```

- Java crea automáticamente un constructor por defecto cuando no existen otros constructores. Inicializa los datos numéricos (int, float,...) a cero, las variables de tipo boolean a false y las referencias a null.

# Constructores

## II – Alternativos

- Un constructor con parámetros se denomina **constructor alternativo**.
- Por ejemplo, un constructor alternativo para la clase Punto sería:

```
public class Punto {  
  
    private double x;  
    private double y;  
  
    public Punto(double _x, double _y) {  
        x=_x;  
        y=_y;  
    }  
}
```

- Ahora para crear objetos Punto tenemos que dar valores a sus coordenadas x e y:

```
Punto p = new Punto(5.0, 47.7);
```

- **OJO:** Si se define una clase con un solo constructor con argumentos y se omite el constructor sin parámetros, ya no será posible utilizar el constructor por defecto.



# Constructores

## III – Sobrecargados

- Es posible **sobrecargar un constructor**, es decir, definir en su clase más de un constructor, cada uno de ellos con una **firma de argumentos distinta** (distinto número de argumentos o argumentos de distintos tipos).
- Los constructores sobrecargados son bastante frecuentes, ya que proporcionan formas alternativas de crear objetos de una clase.
- Sólo un constructor se ejecuta cuando se crea un objeto, con independencia de cuántos constructores se hayan definido.



# Miembros estáticos

- Son los miembros (atributos y métodos) de una clase que pueden ser accedidos sin tener que crear previamente un objeto.
- Se accede a ellos directamente desde la clase.
- Se declaran con la palabra reservada **static**.



# Miembros estáticos

## I – Atributos estáticos

- Un atributo **static** indica que el atributo no pertenece a las instancias de la clase, sino a la propia clase.
- Es decir, que si tenemos varias instancias de una misma clase, cada una de ellas no tendrá una copia de ese atributo, si no que todas compartirán una misma copia del atributo.
- A los atributos estáticos, se les suele llamar **atributos de la clase**.



# Miembros estáticos

## I – Atributos estáticos

- Ejemplo:

```
public class Persona {  
    private String dni;  
    private String nombre;  
    private String apellidos; } Atributos de instancia  
    public static int numPersonas; ← Atributo de clase  
  
    public Persona(String dni, String nombre, String apellidos) {  
        this.dni = dni;  
        this.nombre = nombre;  
        this.apellidos = apellidos;  
        numPersonas++;  
    }  
  
    @Override  
    public String toString() {  
        return "Persona{ " + "dni=" + dni + ", nombre=" + nombre +  
            ", apellidos=" + apellidos + '}';  
    }  
}
```



# Miembros estáticos

## I – Atributos estáticos

```
public static void main(String[] args) {  
    Persona p1=new Persona("12345678A", "Luis", "Pérez");  
    Persona p2=new Persona("98765432B", "Marta", "Cuesta");  
    System.out.println(Persona.numPersonas);  
}
```

- Como vemos, una característica que tienen los atributos de clase, y en general los miembros de clase, es que pueden ser accedidos sin crear un objeto, únicamente con el nombre de la clase.
- ¿Cuál será el resultado del código anterior?



# Miembros estáticos

## II – Métodos estáticos

- Se cargan en memoria en tiempo de compilación y no a medida que se ejecutan las líneas de código del programa, como ocurre con los métodos “normales” o métodos de instancia.
- Para invocar a un método estático no se necesita crear un objeto de su clase, únicamente anteponiendo el nombre de la clase.
- Suelen emplearse para realizar **operaciones comunes a todos los objetos de la clase**.
- No afectan a los estados de los objetos.



# Miembros estáticos

## II – Métodos estáticos

- Añadimos dos métodos a la clase Persona de nuestro ejemplo:

```
public void asignarNombreApellidos(String nombreCompleto) {  
    nombreCompleto=nombreCompleto.trim();  
    int posBlanco=nombreCompleto.indexOf(" ");  
    this.nombre=nombreCompleto.substring(0,posBlanco);  
    this.apellidos=nombreCompleto.substring(posBlanco+1).trim();  
}
```

Resultado diferente en función  
del objeto que lo invoque

```
public static String genero() {  
    return "Humano";  
}
```

Devolverá siempre lo mismo



# Miembros estáticos

## II – Métodos estáticos

- No conviene usar muchos métodos estáticos, ya que aunque se aumenta la rapidez de ejecución de los programas, al cargarse los métodos en memoria en tiempo de compilación y no en tiempo de ejecución, se pierde flexibilidad, no se hace un uso efectivo de la memoria y no se trabaja según los principios de la POO.
- Muchas clases del API de Java disponen de métodos estáticos: **Math** y **Thread** son algunas.



# Miembros estáticos

## III - Reglas

1. Los métodos estáticos no tienen referencia **this**.
  2. Un método estático no puede acceder a miembros que no sean estáticos.
  3. Un método no estático puede acceder a miembros estáticos y no estáticos.
- Como hemos comentado, un miembro **static** existe sin necesidad de instanciar la clase y es compartido por todas las instancias de esa clase.  
Por el contrario, un miembro **no static**, no existe hasta que se cree un objeto de la clase y cada instancia tiene su propia copia del atributo.
  - Esto hace, que desde un método **no static** podamos usar sin problemas los atributos y métodos **static**, pero no al revés. Desde un método **static** no se puede usar o llamar directamente a un atributo o método **no static**.



# Miembros estáticos

## IV - Utilidades

- Definir constantes **final**. Si no las hacemos **static**, cada instancia de la clase tiene su propia copia de la constante, lo que es un desperdicio de memoria.

```
public static final double PI = 3.1416;
```

- Definir **atributos** que interese que sean **compartidos** entre todas las instancias de una clase. Por ejemplo un contador de objetos.
- Definir métodos que no usen atributos de la clase. Suelen ser **métodos de "utilidad"**.



# Excepciones



- Una **excepción** es una **condición anormal** que ocurre durante la ejecución de un programa.
- Las excepciones “deben” ser manejadas (tratadas) por el programador, a proporcionando un código llamado **manejador de excepciones**.
- Ejemplos típicos de excepciones en Java :
  - Operaciones aritméticas incorrectas, como por ejemplo dividir entre cero → **ArithmeticException**.
  - Uso de **referencias nulas**, es decir, punteros a objetos que no existen → **NullPointerException**
  - Acceso a **posiciones inválidas de un array** → **IndexOutOfBoundsException**
  - Acceso a **ficheros inexistentes** → **FileNotFoundException** ó **IOException**.

# Excepciones

- El manejador de excepciones, intentará que el programa se recupere del error, **intentándolo** de nuevo o simplemente **continuando**.
- De esta forma, podemos decir que hay dos modelos básicos en la teoría de manejo de excepciones, la **terminación** y la **reanudación**.
- En la **terminación** (que es lo que soporta Java) se asume que el error es tan crítico que no hay forma de volver atrás para resolver la situación donde se produjo la excepción.
- En la **reanudación**, se espera que el manejador de excepciones haga algo para rectificar la situación, y después se vuelve a ejecutar el método que causó el error.
  - En este caso, la excepción es más como una llamada a un método, es decir, es mejor llamar a un método que solucione el problema que lanzar una excepción.
  - Alternativamente, se puede ubicar el bloque **try** dentro de un bucle **while**.



# Excepciones

- Si aparece una excepción en un método existen 2 posibilidades:
  - a) **Capturar** la excepción en ese mismo método y tomar las acciones necesarias para manejarla.
  - b) **No capturar** la excepción en ese método y pasarla al método superior,
- Si nadie se “responsabiliza” del manejo de la excepción ésta llegará al **main()**, el cual puede a su vez hacer 2 cosas, ó manejar él mismo la excepción, ó pasarla al intérprete de Java, el cual finalizará la ejecución del programa bruscamente y mostrará por pantalla la información relacionada con dicha excepción (nombre, lugar donde ha aparecido, etc...).
- **Esto último hay que evitarlo**, ya que todo programador debe controlar (manejar) los posibles errores que puedan surgir en la ejecución de un programa, no dejando en ningún caso que sea Java el que nos informe de ellos en tiempo de ejecución.



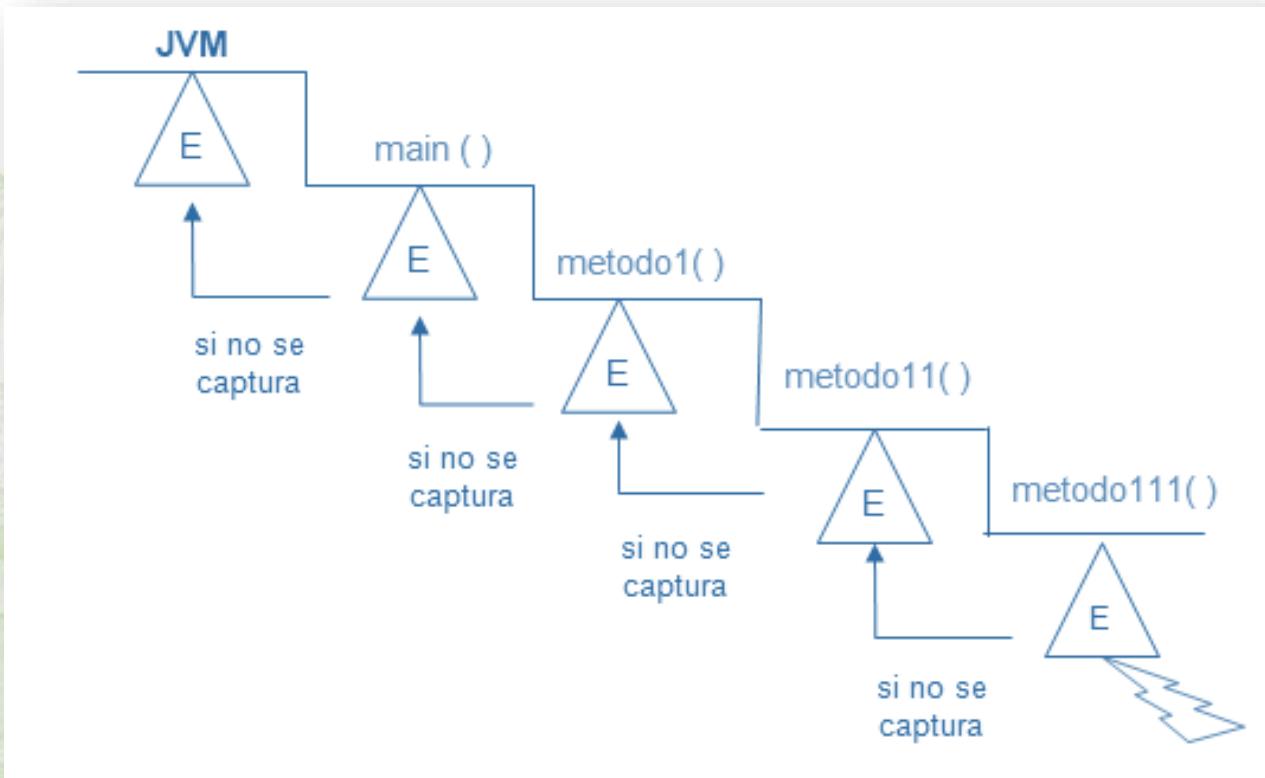
# Excepciones

- Así, el mecanismo de excepciones permite la:
  1. Detección de errores y su posible recuperación.
  2. Limpieza y salida elegante en el caso de errores no manejados.
  3. Propagación de errores no manejados.



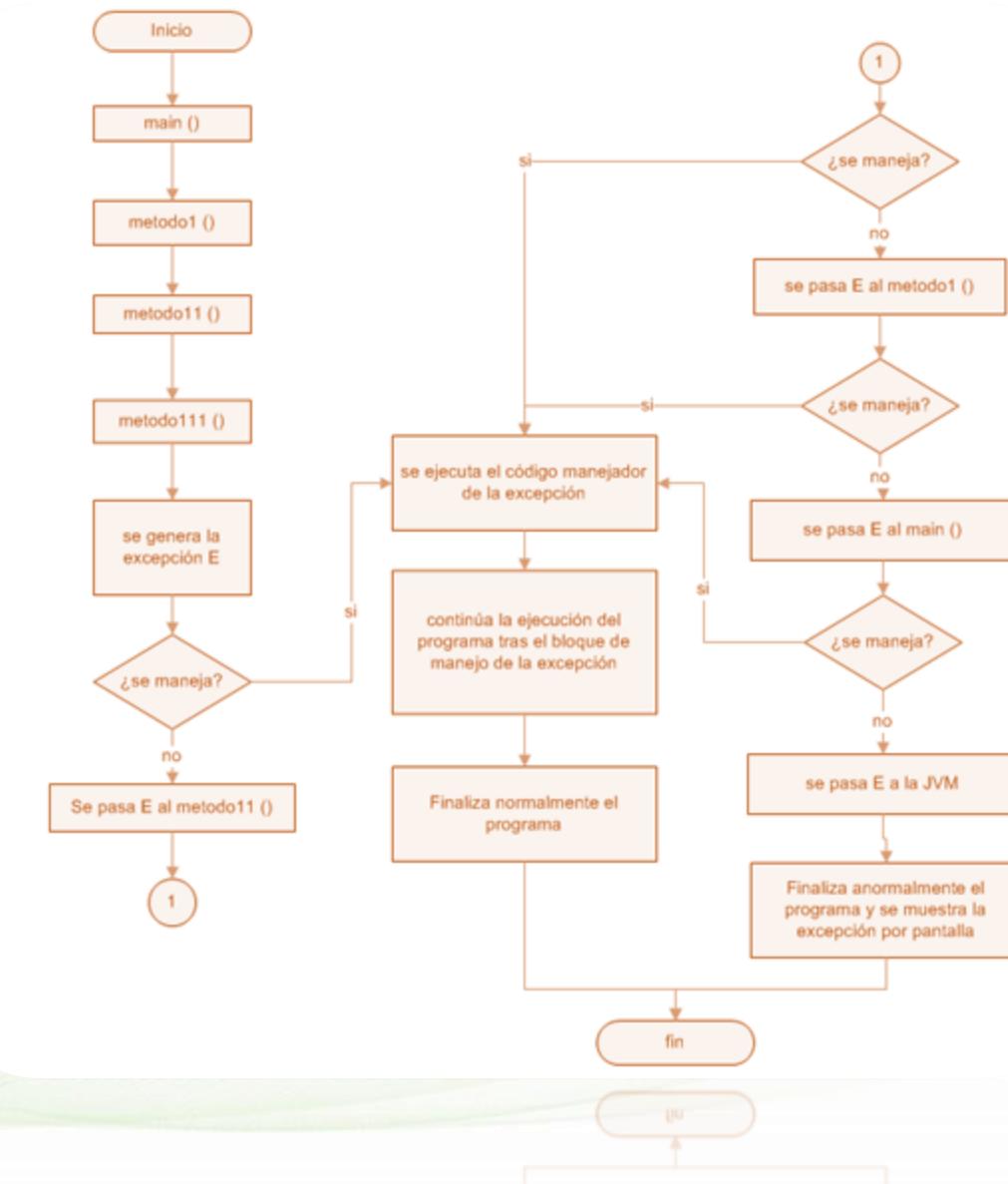
# Excepciones

- Supongamos un programa con una serie de métodos enlazados y en el último se genera una excepción:



# Excepciones

- Diagrama de flujo:



# Excepciones

## I – Bloque try/catch/finally

- En Java para manejar excepciones se utilizan los bloques de sentencias **try – catch - finally**.
- **Sintaxis:**

```
try {  
    // Código que podría generar excepciones  
} catch(Tipo1 id1) {  
    // Manejo de excepciones de Tipo1  
} catch(Tipo2 id2) {  
    // Manejo de excepciones de Tipo2  
} catch(Tipo3 id3) {  
    // Manejo de excepciones de Tipo3  
}
```



# Excepciones

## I – Bloque try/catch/finally

- El **bloque try** contendrá el código que puede generar excepciones.
- El **bloque catch** es el manejador de excepciones, ya que contiene el código que se ejecutará en caso de que aparezcan excepciones.
  - En caso de que puedan surgir varias excepciones pueden incluirse varios bloques **catch**, uno por cada tipo de excepción.
  - Cada cláusula **catch** es semejante a un método que toma un argumento de un tipo en particular.
  - Si se lanza una excepción el mecanismo de gestión de excepciones trata de cazar **el primer manejador (catch)** con un argumento que coincida con el tipo de excepción.
  - Sólo se ejecuta **una cláusula catch**.



# Excepciones

## I – Bloque try/catch/finally

- Y el **bloque opcional finally** contendrá sentencias que se ejecutarán siempre, haya o no excepción.
  - Se suele utilizar para cerrar conexiones, liberar recursos, cerrar ficheros, etc.



# Excepciones

## I – Bloque try/catch/finally

- Ejemplo:

```
1. int x = (int) (Math.random() * 5);  
2. int y = (int) (Math.random() * 10);  
3. int z[] = new int[5];  
4. try {  
5.     System.out.println("x/y da " + (x/y));  
6.     System.out.println("y es " + y + " z[y] es " + z[y]);  
7. } catch (ArithmaticException e) {  
8.     System.out.println("Problema aritmético: " + e);  
9. } catch (ArrayIndexOutOfBoundsException e) {  
10.    System.out.println("Problema de índice: " + e);  
11. } finally {  
12.    System.out.println("Bloque finally");  
13.}
```

# Excepciones

## II – Lanzamiento

- Sentencia **throw**
  - Se pueden **lanzar excepciones por nosotros mismos** mediante la sentencia **throw**.
  - El proceso para lanzar una excepción mediante código es muy sencillo:

```
throw new <tipo_excepción>([<mensaje>]);
```



# Excepciones

## II – Lanzamiento

- Si un método lanza una excepción mediante **throw** Java **OBLIGA** a manejar la excepción de alguna forma.
  - a) Mediante un bloque **try-catch** en el mismo método.
  - b) Podemos hacer que la excepción no sea tratada en el propio método, sino que **sea propagada al método superior** (llamante), es decir, que el bloque **try-catch** sea codificado en el método superior.
- En estos casos, hay que indicar al compilador que **el método puede propagar** una excepción (o varias), poniendo la sentencia **throws** en la cabecera del método seguida del tipo o tipos de excepción que puede lanzar.

**throws <tipo\_excepción1> [<tipo\_excepción2>] [...];**



# Excepciones

## II – Lanzamiento

- Ejemplo:

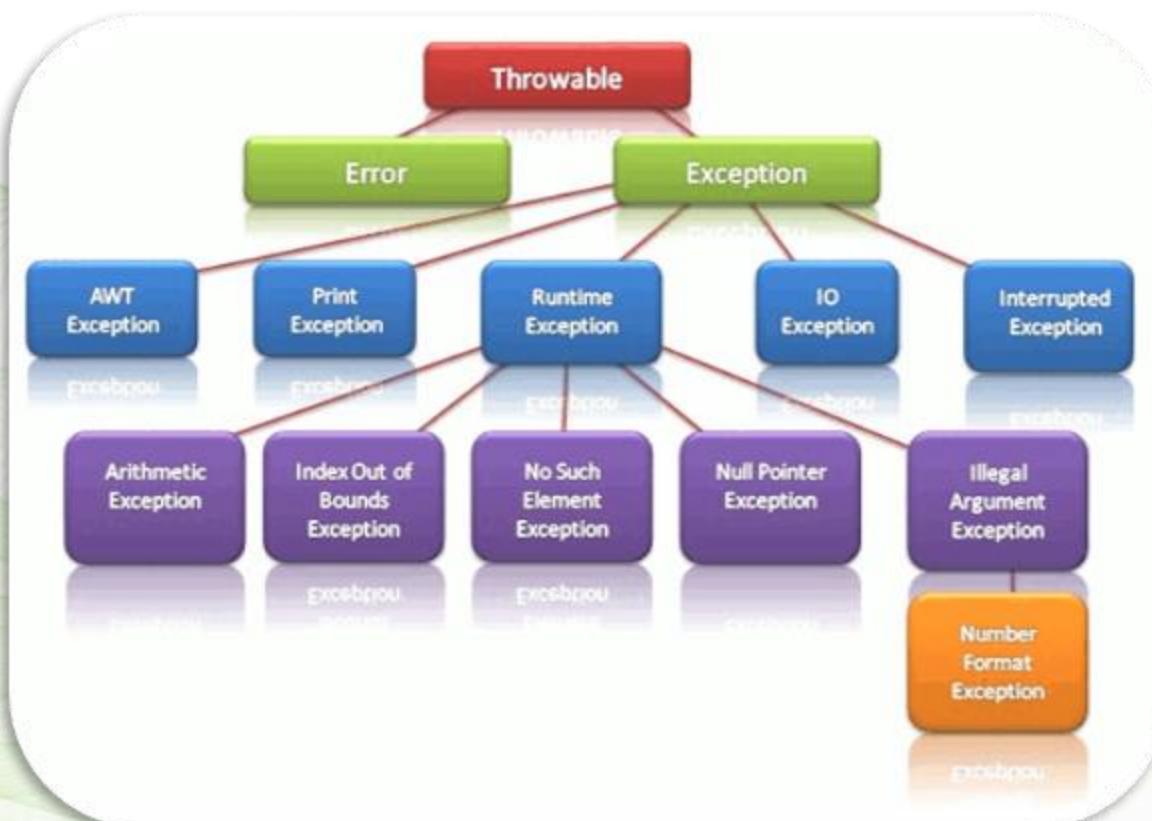
```
public static FileReader abrirFichero(String fichero)
    throws FileNotFoundException {
    FileReader fr = new FileReader(fichero);
    return fr;
}
```



# Excepciones

## III – Tipos

- Todas estas clases pertenecen al paquete **java.lang**.



# Excepciones

## III – Tipos

- Todos los tipos de excepciones heredan de **Throwable**.
- Dentro de esta categoría “padre” existen 3 categorías de excepciones:
  1. Excepciones que es **obligatorio manejar**. Heredan de **Exception**.
  2. Excepciones que **no es obligatorio manejar**.  
Se subdividen en:
    - Excepciones en tiempo de ejecución. Heredan de **RuntimeException**.
    - Errores. Heredan de **Error**.



# Excepciones

## III – Tipos

- Las **excepciones obligadas de manejar (Exception)** normalmente son problemas relacionados con el entorno, tales como errores de usuario o problemas de entrada/salida. Ninguno de ellos son problemas de codificación, aunque *Java fuerza a escribir código* en previsión de que ocurran.
- Las **excepciones en tiempo de ejecución (RuntimeException)** no tienen por qué manejarse aunque puede hacerse.

No obstante como estas excepciones representan errores de programación tales como accesos fuera de los límites de un array o utilización de referencias nulas **no deberían manejarse** siendo un mal estilo de programación el hacerlo. Es decir, no hay que hacer un **try-catch** cada vez que utilizamos una referencia a un objeto.

- Por último **los errores (Error)** describen problemas que son muy inusuales, como por ejemplo problemas de falta de memoria.

Este tipo de excepciones por lo inusual no son necesarias de manejar.



# Excepciones

## III – Tipos

- A la hora de manejar una excepción, podemos utilizar unos métodos comunes a todas las excepciones:
  - **String getMessage()**. Obtiene el mensaje descriptivo de la excepción.
  - **String toString()**. Obtiene una cadena con la situación de la excepción. Suele indicar la clase de excepción y el texto de `getMessage()`.
  - **void printStackTrace()**. Escribe la información de la pila de llamadas. El resultado es el mismo mensaje que muestra la máquina virtual de Java cuando no se controla la excepción.

```
try{  
    ....  
} catch (IOException e){  
    System.out.println(e.getMessage());  
    System.out.println(e.toString());  
    e.printStackTrace();  
}
```



# Excepciones

## IV – Personalizadas

- Podemos **crear nuestras propias excepciones** para indicar un error especial que no fue previsto cuando se creó la jerarquía de excepciones de Java.
- Para crear nuestra propia excepción crearemos una clase que modele el nuevo tipo, la cual **heredará** de un tipo de excepción existente, preferentemente uno cercano al significado de la nueva excepción, sin embargo, como a menudo esto no es posible, heredará de la clase base **Exception**.
- La forma más trivial de crear un nuevo tipo de excepción es simplemente dejar que el compilador cree el constructor por defecto, de forma que prácticamente no haya que escribir ningún código.



# Excepciones

## IV – Personalizadas

- Ejemplo:

```
public class ExpcionSencilla extends Exception {  
    //Constructor por defecto  
}  
  
public class EjemploExpcionSencilla {  
  
    public static void metodo1() throws ExpcionSencilla {  
        System.out.println("Lanzando ExpcionSencilla");  
        throw new ExpcionSencilla();  
    }  
  
    public static void main(String[] args) {  
        try {  
            metodo1();  
        } catch (ExpcionSencilla e) {  
            System.out.println("Capturada");  
        }  
    }  
}
```



# Excepciones

## IV – Personalizadas

- Ampliamos el ejemplo anterior con un constructor parametrizado:

```
public class ExpcionSencilla extends Exception {  
    //constructor por defecto  
    public ExpcionSencilla() {}  
  
    //constructor parametrizado  
    public ExpcionSencilla(String msg) {  
        super(msg);  
    }  
}
```



# Excepciones

## IV – Personalizadas

- Ahora podemos lanzar excepciones de dos formas:

```
public class EjemploExcepcionSencilla {  
  
    public static void metodo1() throws ExcepcionSencilla {  
        System.out.println("Lanzando ExcepcionSencilla");  
        throw new ExcepcionSencilla();  
    }  
  
    public static void metodo2() throws ExcepcionSencilla {  
        throw new ExcepcionSencilla("Lanzada desde metodo2");  
    }  
}
```



# Excepciones

## IV – Personalizadas

- Las capturamos:

```
public static void main(String[] args) {  
  
    try {  
        metodo1();  
    } catch (ExceptionSencilla e) {  
        System.out.println("Capturada");  
    }  
  
    try {  
        metodo2();  
    } catch (ExceptionSencilla e) {  
        System.out.println(e.getMessage());  
    }  
}
```

Se muestra el propio mensaje de la excepción





# Actividades



Actividades Tema5

gf

Centro de Enseñanza  
Gregorio Fernández