1. Introducción

Las características fundamentales de la POO son: Abstracción, Encapsulación, Herencia y Polimorfismo. Las dos primeras las hemos desarrollado en el tema anterior. Ahora, profundizaremos en las características avanzadas de la POO.

2. Herencia

2.1. Definiciones

Herencia: propiedad que nos permite crear nuevas clases a partir de clases existentes, conservando propiedades de la clase original e incorporando otras nuevas.

La nueva clase obtenida se llama **clase derivada** o hija y la clase a partir de la cual se crea se llama **clase base** o padre.

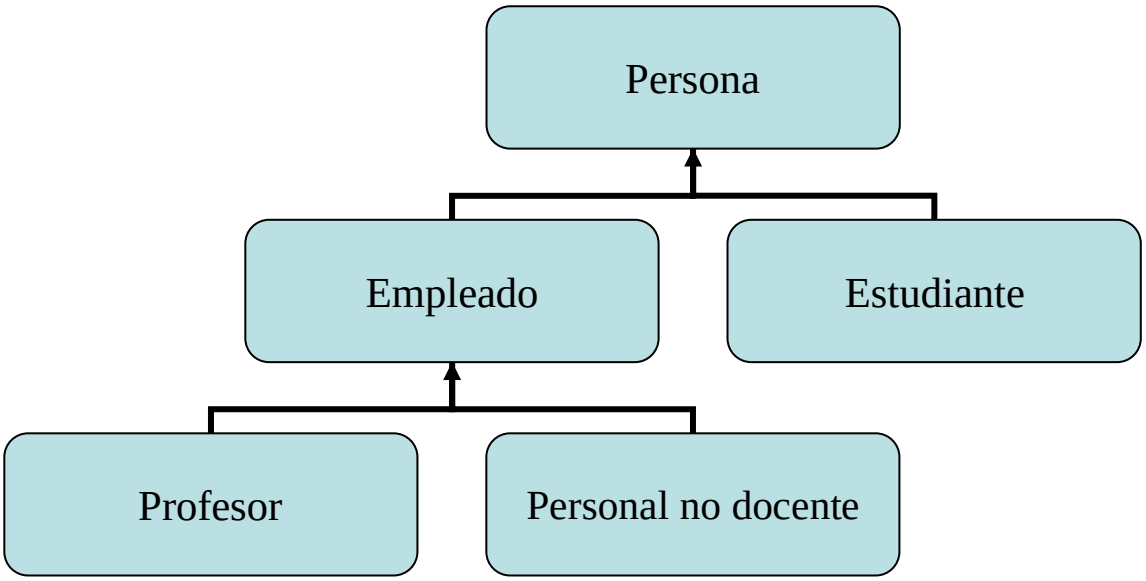
Con la herencia, las clases se clasifican en una jerarquía estricta. Cada clase tiene una superclase (de la cual hereda) y una o varias subclases (que derivan de ella).

Puede ocurrir que tengamos también derivación múltiple en la que cada clase tiene más de una superclase (no soportada directamente por JAVA).

Ventajas de la jerarquía de clases:

- Cada vez que se crea un objeto derivado se crea un solo objeto que contiene toda la información heredada.
- El programa será más flexible a cambios y ampliaciones.

Gráficamente se puede representar la herencia entre clases de la siguiente manera:



2.2. Ejemplo de herencia en JAVA.

Ejemplo: Tendremos una clase A de la cual se deriva la clase B.

```
package ejemplo_herencia;
public class A {
   protected int a;
   public A()
   {
      a=9;
   }

   public A(int n) {
      a=n;
   }

   public int dev_valor() {
      return a;
   }

   public void cambia_valor(int nuevo) {
      a=nuevo;
   }
}
```

```
public class B extends A {

   private int b;

    public B(){
      a=12;
      b=18;
   }
    public B(int nb) {
      b=18;
   }
   public B(int na, int nb){
      super(na);
      b=nb;
   }
    public int dev_valorb() {
      return b;
   }
   public void cambiar_valorb(int nuevo) {
      b=nuevo;
   }


}
```

3. Definición de la herencia en Java

3.1. Subclases y herencia clásica.

En la herencia clásica tendremos una clase base y una o varias derivadas que heredan de la base tanto atributos como métodos. Podremos definir objetos tanto para la clase base como la derivada. Si en la derivada aparecen métodos que tienen el mismo nombre que en la clase base, los de la clase hija sobrescriben (override) los de la clase base. De modo que una llamada a estos métodos por parte de un objeto hijo llamarán al método definido para la clase derivada y no el de la clase base.

La clase derivada se define de la siguiente manera:

public class *derivada* **extends** *base* {

…

}

Con esto la clase derivada incorporará atributos y métodos de la clase base.

JAVA no permite la derivación múltiple, es decir, una subclase no puede heredar de más de un padre.

Una subclase o clase derivada puede, a su vez, se la superclase o clase base de otra clase, apareciendo la jerarquía de clases.

- Control de acceso a los miembros de una clase.

Puede ser accedido por:	**Miembro declarado como**			
	Private		***Protected***	***Public***
Su misma clase	Sí		Sí	Sí
Cualquier clase o subclase del mismo paquete	No		Sí	Sí
Cualquier clase de otro paquete	No		No	Sí
Cualquier subclase de otro paquete	No		Sí	Sí

- ¿Qué miembros hereda una subclase?
  - Una subclase hereda todos los miembros de la superclase, excepto los constructores, lo que no significa que tenga acceso a todos esos miembros; dependerá de cómo estén definidos en la superclase.
  - Una subclase puede añadir sus propios miembros. Si alguno coincide con los de la superclase, los de esta última quedarán ocultos, prevaleciendo los de la clase hija.
  - Los miembros heredados pueden ser a su vez heredados por otras subclase (siempre que las clausulas de definición lo permitan).

- Miembros con el mismo nombre:
  - Atributos: El atributo que queda accesible y manejable es el de la clase derivada. Si aún así, necesitamos acceder desde la clase derivada al atributo de la clase base usaremos la instrucción: *super.atributo_clase_base.*

  - Métodos: Cuando en una subclase se redefine un método que estaba definido en la clase base se sobrecarga, a esto se le llama *override*.

- Diferencias entre métodos heredados y atributos heredados. Un método no se puede redefinir menos restrictivo, es decir, se debe seguir el orden: *private*, *protected* y *public*.

  Es decir: si el método de la superclase es:

  - *Protected*: el de la subclase puede ser protected o public, no prívate.
  - *Public*: el de la subclase sólo puede ser public.
  - *Prívate*: no se puede redefinir ya que sólo es accesible por su propia clase.

- Para acceder, de todas maneras, al método de la superclase se utiliza la clausula '*super'* igual que con los atributos.

  Ejemplo:
  Tenemos la siguiente clase base:

```
public class Base
{
   protected int dato_base;
   protected int otro_dato;

   public Base() {dato_base=18;otro_dato = 6;}
   public Base(int db) {dato_base = db;}

   @Override
   protected void finalize()
   {
      System.out.println("Destruyendo la clase base.");
   }

   public int dev_dato_base() {return dato_base;}
   public void cambia_base(int d) {dato_base = d;}
   //public abstract void opera();
   public int dev_otro_dato()
      {
         return otro_dato;
      }
}
```

  Y la siguiente clase derivada:

```
public class Derivada extends Base
{
   private int dato_derivado;
   private int otro_dato; //Se sobrescribe el atributo de la clase base.

   public Derivada()
   {
      dato_derivado = 9;
      otro_dato = 12;
   }

   public Derivada(int dd, int od, int db)
   {
      super(db); //Llamamos al constructor con parámetros de la clase base.
      dato_derivado = dd;
      otro_dato = od;
   }

   @Override
   protected void finalize()
   {
      System.out.println("Destruyendo la clase derivada.");
      super.finalize();
   }
```

```
    public int dev_dato_derivado() {return dato_derivado;}
    public void cambia_derivado(int d) {dato_derivado = d;}
    public void opera() {dato_base = dato_base * dato_derivado;}
    @Override
    public int dev_otro_dato()
        {
            return otro_dato;
            //return super.otro_dato;  --> Esto accedería al atributo de la clase base.
            //return super.dev_otro_dato();  --> accede al método de la clase derivada.
        }
}
```

3.2. Superclases o clases abstractas.

Cuando se diseña una clase para que sea genérica y sirva de base para otras clases que heredan de ella atributos y métodos, comunes a todas las subclases, esta clase genérica se denomina abstracta y no podremos definir objetos de esta clase.

Para especificar una clase abstracta en JAVA se usa la palabra reservada '*abstract'*. La definición genérica es:

public abstract class loquesea { ….

}

Las clase abastractas tienen las siguientes características:

- No podemos definir objetos de ellas.
- Pueden contener métodos abstractos (sin cuerpo) y no abstractos (con cuerpo).

Características de los métodos abstractos:

- No tienen cuerpo.
- En cuanto una clase tiene un método de estos, la clase debe ser automáticamente abstracta.
- Se deben redefinir obligatoriamente en las clases derivadas, donde sí tendrán cuerpo y se adaptarán a las necesidades de cada subclase

Ejemplo de clase abstracta: supongamos que tenemos que guardar información sobre instrumentos musicales. Queremos que todos hereden de una clase común *instrumento* y que implementen unos métodos determinados:

```
public abstract  class Instrumento {
    public void metodo_Instrumento(){
        //Este método estará compartido/heredado por todos los hijos polimórficos
        System.out.print("Esto está definido en el padre. Y todos los hijos polimórficos lo pueden usar.");
    }
    public abstract void afinar();
```

```
    public abstract void tocar(); //Al tener métodos abstractos la clase se convierte en abstracta. En el
                //main no podremos definir objetos de esta clase.
}
```

Los hijos se definen:

```
public class Viento extends Instrumento {
    public void metodo_viento(){
        //Este método no es accesible via Polimorfismo
    }
   @Override
     public void afinar(){
        System.out.println("Afinando el viento");
    }
  @Override
    public void tocar(){
        System.out.println("Tocando el viento");
    }
}
```

Y por ejemplo:

```
public class Cuerda extends Instrumento {
    public void metodo_cuerda(){
        //Este método no es accesible via Polimorfismo
    }
 public void afinar(){
        System.out.println("Afinando el cuerda");
    }
 public void tocar(){
        System.out.println("Tocando el cuerda");
    }
}
```

Se puede observar como los métodos afinar y tocar no tiene cuerpo en la clase base y sí en las derivadas.

4. Otras características de la herencia en Java

4.1. Constructores.

Cuando se definen constructores para la clase ocurre lo siguiente: se ejecutan los constructores de arriba abajo; el último en ejecutarse será el de la subclase.

Sin embargo, cuando se hayan definido constructores con parámetros se pueden invocar los constructores de la clase base de forma explícita de la siguiente manera:

```
public class Papa {
   protected int a;
   public Papa(){
      a=0;
   }
   public Papa(int num){
      a=num;
   }
   public int dev_Papa(){
      return a;
   }
}

public class Hija extends Papa {
   private int b;
   public Hija(){
      b=0;
   }
   public Hija(int num){
      b=num;
   }
   public Hija(int num, int num2){
      super(num);
      b=num2;
   }
}
```

4.2. Destructores.

Si definimos un método *finalize* en la subclase, lo correcto es crear otro en la superclase. La manera de invocarlo es, al contrario que los constructores, de abajo a arriba. Es decir, se invocaría primero el *finalize* de las subclase y por último el de las superclases.

El método de la clase derivada sería:

```
protected void finalize()
   {
      System.out.println("Destruyendo la clase derivada.");
      super.finalize();
   }
```

4.3. Otras características.

Dos características importantes de la herencia que nos quedan por ver son:

- Si queremos que un método de una superclase no se pueda sobrecargar en la subclase, se debe definir con la clausula *final*. Esto no quiere decir que no se pueda usar en la subclase, no, al contrario, sí que se puede usar en la subclase, lo que no se podrá hacer es sobrecargarla en la clase derivada.

  Ejemplo: El siguiente método definido en una clase base, no se podrá redefinir en una clase derivada.

  ```
  public final int dev_dato_base() {return dato_base;}
  ```

- Los métodos *static* no se heredan.

5. Polimorfismo.

El polimorfismo tiene que ver con la relación que se establece entre la llamada a un método y el código que efectivamente se ejecuta. A esta relación se le llama vinculación o *binding*.

Puede ser:

- **Temprana**: se conocer en tiempo de compilación (sobrecarga).
- **Tardía**: se conoce en tiempo de ejecución (*override* o redefinición).

  Esta vinculación permite que, si tenemos un método redefinido en carias clases, sea el tipo de objeto y no la referencia en sí la que determine qué método se ejecuta.

  Según lo expuesto, podremos tener una matriz o vector en el que cada elemento del vector sea un objeto distinto.

  Ejemplo:

  Tomando como base el ejemplo de los instrumentos de la sección 2.2, podemos tener el siguiente *main*:

```
Instrumento orquesta [] = new Instrumento[4];
int i;
orquesta[0]=new Cuerda();        orquesta[1]=new Viento();
orquesta[2]=new Teclados();        orquesta[3]=new Percusion();
for(i=0; i<4;i++)
{   orquesta[i].afinar(); //--> Llamaría cada cual el suyo.
   orquesta[i].metodo_Instrumento();//--> Todos llaman al método padre.
}
```

6. Interfaces.

Lleva un paso más allá el concepto de clase abstracta. En JAVA una interfaz es una clase abstracta pura en la que:

- Todos los métodos son abstractos y públicos. No se implementa ninguno.
- Los atributos (caso de que los haya) son estáticos, públicos y finales.
- No se pueden crear objetos de las interfaces.

Declaración:

```
Interface nombre_interfaz { … }
```

Uso: para que una clase use o herede de una interfaz se usa la partícula implements. En todo lo demás es herencia y polimorfismo.

```
class la_que_sea implements nombre_interfaz { ….}
```

Se pueden implementar varias interfaces.

Ejemplo:
Si necesitáramos, por un lado, una clase base de la que tenemos que definir objetos y además necesitamos que todos tengan una serie de métodos obligatoriamente podemos definir:

La interfaz:

```
public interface Metodos {
   public abstract void afinar();
   public abstract void tocar();
}
```

La clase base:

```
public class Instrumento  implements Metodos { //Ya no es abstracta por lo que podemos definer
                        //objetos de ella
   public void metodo_Instrumento(){
      //Este método estará compartido/heredado por todos los hijos polimórficos
      System.out.print("Esto está definido en el padre. Y todos los hijos polimórficos lo pueden usar.");
   }
}
```

La clase derivada:

```
public class Viento extends Instrumento implements Metodos {

    public void metodo_viento(){
       //Este método no es accesible via Polimorfismo
    }
    @Override
    public void afinar(){
       System.out.println("Afinando el viento");
    }
    @Override
    public void tocar(){
       System.out.println("Tocando el viento");
    }
}
```

Resumiendo, una interfaz sirve para:

- Asegurarnos que las clases que la implementan tengan todos una serie de métodos.
- Realizar una especie de herencia múltiple.

Agrupar una serie de valores constantes, ya que todos sus atributos son estáticos y finales.

7. Clases internas o anidadas.

Una clase anidada o interna es una clase definida dentro de otra. Esta clase la definiremos con las mismas reglas que el resto de clases. Pueden ser incluso clases estáticas.
La diferencia es que esta clase al estar dentro de otra tendrá acceso a todo lo de la clase que la contiene aunque sean privados.
Estas clases internas surgen cuando tenemos que algún atributo privado de una clase es demasiado complejo para ser un atributo clásico y no tenemos necesidad de crear objetos de esa clase fuera de la clase contenedora.

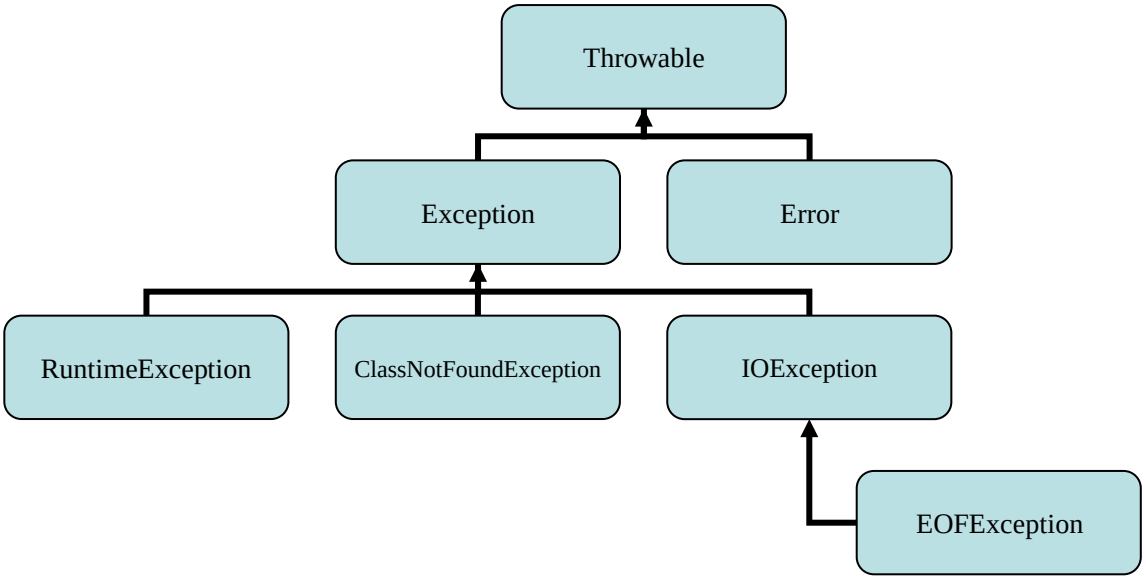
Ejemplo:

```
public class Pilas {


   //Clase privada interna.
   private class T_Nodo {
      private  String ele;
      private  T_Nodo sig;

      public T_Nodo(){
         ele="";sig=null;
      }
      public T_Nodo(String e){
         ele=new String(e);
         sig=null;
      }
   }


   //----------------- Atributos de la pila ----------------------
   private T_Nodo cima;


…
```

8. Excepciones.

Una excepción se define como un error que se produce en el programa y que es recogido por el propio programa.
En JAVA son objetos de clases derivadas de la clase Throwable definida en java.lang. Cuando ocurre algún error en ejecución se lanza una excepción que crea automáticamente un objeto de la clase correspondiente.

Jerarquía de clase de las excepciones:



Podemos observar que hay, principalmente, dos tipos de excepciones:

- Un objeto *Error* se crea cuando ha ocurrido un problema serio. Este tipo de errores involucran a la máquina virtual por lo que una aplicación normal no puede manejar esta excepción.
- Un objeto *Exception* cubre las excepciones que una aplicación convencional manipula.

Manejar excepciones.

Cuando ocurre un error se debe capturar dicho error. Para realizar esto se siguen dos pasos:

- El código delicado lo encerramos en un bloque
  try {...}
- Escribimos, al menos, un capturador con
  catch(Tipo_Excepcion e) { ... }

El orden de captura de las excepciones será de subclases hacía las superclases.

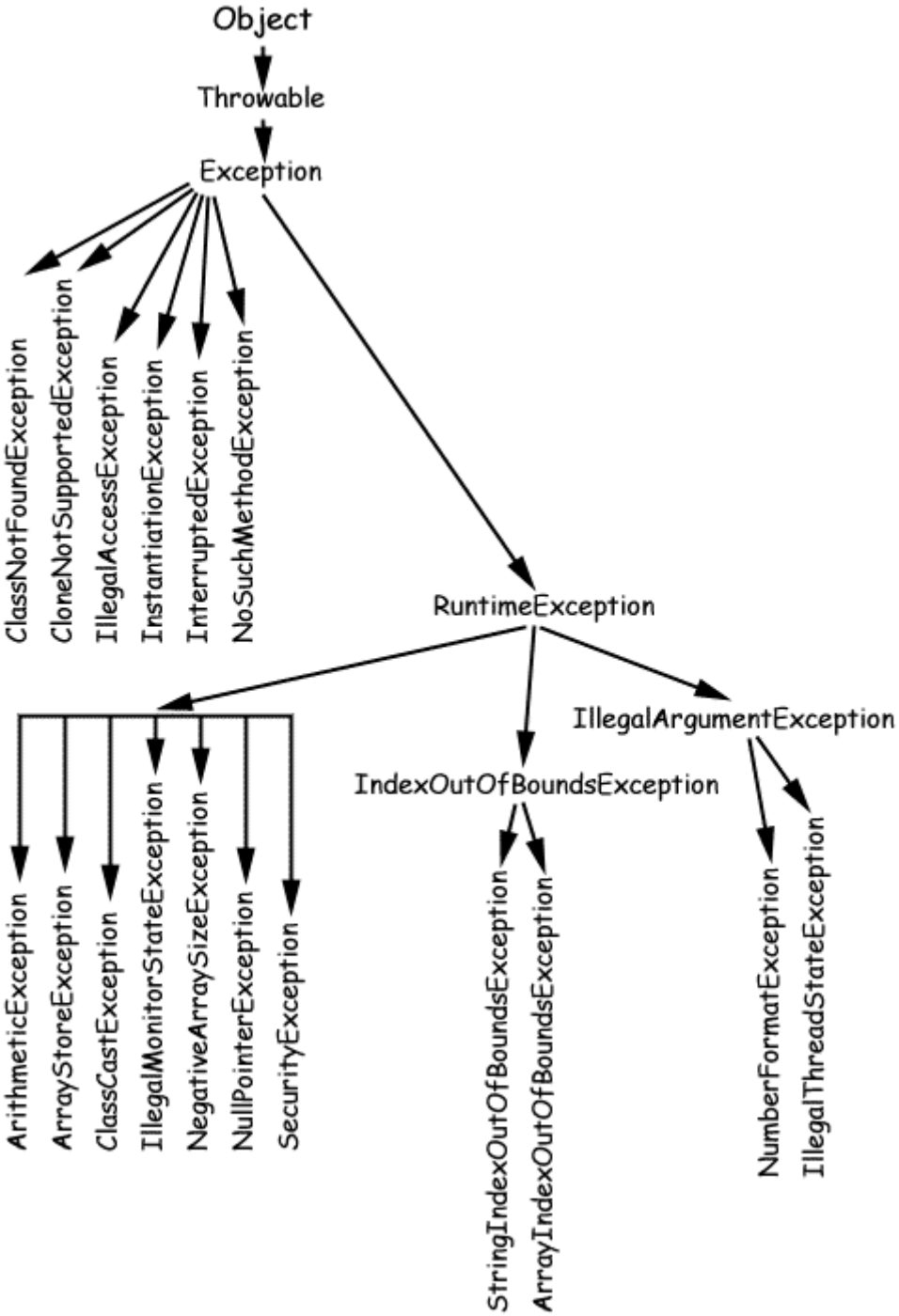
Ejemplo:

```
public static void main(String[] args) throws IOException
   {boolean salir=true;
    BufferedReader flujoE = new BufferedReader(new InputStreamReader(System.in));

     do {
        salir=true;

     try {
      A obj = null;  obj = new A();
      A[] v=new A[4];
      int n = 10,m=0;
```

```
      //******* Excepciones lanzadas automáticamente **************
      //a) Error: 'NullPointerException'
obj.muestra();
obj = new A();
obj.muestra();
      //b)Error: ArrayIndexOutOfBoundsException
          v[7] = new A();
      //c)Si introducimos letras --> Error: NumberFormatException
      System.out.print("Dame un numero: ");
      n = Integer.parseInt(flujoE.readLine());
     }

    //Se colocan los catch en orden descendente por la jerarquía de la herencia.

           catch(RuntimeException ex)
    {
       if (ex.getMessage().equals("7")) //Vector fuera de rango
          System.out.println("Vector mal utilizado");
       else System.out.println(ex.getMessage());

       salir = false;
    }
    catch(Exception ex)
    {//Aquí se recogerán otras excepciones que no sean 'RuntimeException'
       //por ejemplo: 'IOException' / 'EOFException' etc.
       salir = false;
    }
   }while(!salir);
  }
```

Listado de excepciones.

Las excepciones principales que manejaremos serán:



Crear y lanzar excepciones.

En algunos casos es muy interesante el uso de excepciones para el tratamiento de errores. En primer lugar relegamos el tratamiento de los errores a la misma parte del código y segundo podemos unificar el mismo tratamiento para el mismo tipo de error. Es un tipo de solución muy utilizado empresarialmente.

- Declaración de excepciones.
  Para poder utilizar excepciones personalizadas, lo primero será definir una clase que herede de la superclase *Exception*.

  ```
  public class Mis_Excepciones extends Exception
  ```

```
{

   public Mis_Excepciones() {}
   public Mis_Excepciones(String men) {super(men);}

}
```

- Capturar la excepción.
  Tendremos que colocar en nuestro código algo como:
  catch(Mis_Excepciones e) {…}

- Lanzamiento.
  Tendremos que lanzar nuestro error cuando creamos conveniente.
  throw new Mis_Excepciones(“Mensaje personalizado”);

```
public void operar(int b) throws Mis_Excepciones
   {
      if (b==0) throw new Mis_Excepciones("Nuestro mensaje de error.");
      entero = entero / b;
   }
```

El uso de métodos proporciona robustez a nuestro código.