CURSO DE JAVA

PROGRAMACIÓN ORIENTADA A OBJETOS EN JAVA

ÍNDICE

I. CARACTERÍSTICAS POO

- Facilidad de diseño y relación con el mundo real (UML)
- Reusabilidad y facilidad de mantenimiento
- Sistemas más complejos
  - Abstracción
  - Trabajo en equipo
- Del lenguaje máquina hacia el mundo real
- Resuelve problemas complicados. No está pensado para tareas sencillas

2.1 ELEMENTOS. CLASES

- Son la plantilla a partir de la cual se crean los objetos
- Formadas por:
  - Nombre
  - Atributos
  - Métodos
- En general, interesa que los atributos no se puedan tocar directamente desde fuera de la clase

  → privados
- Los métodos son la forma de comunicarse con la clase para pedirle que haga cosas (servicios)

  → púúblicos o privados
- *Los métodos get y set "hacen trampa"*

2.2 OBJETOS

- Son la parte ejecutable de la programación orientada a objetos
- Se manejan a través de variables
- Pertenecen a una clase

3.1 POO EN JAVA. DECLARACIÓN DE CLASES

- Hay que declarar la visibilidad de la clase, sus atributos y sus métodos
- En cada fichero
- una clase pública con el mismo nombre del fichero
- 0..n clases privadas

```
public class Circulo
{
    private int centro_x, centro_y, radio;
    public void paint (Graphics g)
    {
        g.drawCircle (centro_x, centro_y,
        radio, Color.GREEN);
    }
 }
```

3.2 USO DE OBJETOS

- Primero es necesario declarar una variable perteneciente a la clase:

```
Circulo mi_circulo;
```

- Después hay que crear el objeto:

```
mi_circulo = new Circulo();
```

- Ahora ya se puede llamar a los métodos del objeto:

```
mi_circulo.paint();
```

3.3 CONSTRUCTOR

- Método especial que indica lo que se hace cuando se crea un objeto
- Tiene el mismo nombre que la clase
- Pueden existir varios constructores con distintos parámetros
- Si no declaramos ninguno se usa el constructor por defecto
- Si declaramos alguno es obligatorio declarar también el constructor por defecto

```
public class Circulo
{
    int c_x, c_y, radio;
    Circulo (int x, int y) {
        c_x = x;
        c_y = y;
    }
    Circulo (int x, int y,int r) {
        c_x = x;
        c_y = y;
        radio = r;
    }
    Circulo()
    {
        c_x=c_y=radio=1;
    }
}
```

3.4 OBJETO THIS

- La palabra reservada this sirve para que un objeto haga referencia a sí mismo
- Usos:
  - Un objeto se pasa a sí mismo como parámetro al llamar a un método de otro objeto
  - Especificar que un objeto utiliza sus métodos o sus atributos: no es obligatorio, pero a veces es necesario

```
public class Circulo {
        public void paint () {
                Ventana.paint(this);
        }
}

public class Circulo {
        int x,y,r;
        Circulo(int x, int y, int r) {
                this.x = x;
                this.y = y;
                this.r = r;
        }
}
```

3.5 MÉTODOS

- De manera general, la declaración de un método en java es de la forma

```
modo_acceso modificadores tipo_retorno nombre_metodo (argumentos) {
        //Cuerpo del método
}
```

- Donde
  - modo_acceso: **public, private, protected (por defecto package-privated)**
    - No es obligatorio especificarlo, pero es conveniente
  - Modificadores: **static, abstract, final, native, synchronized**
    - No es obligatorio usarlos
  - El tipo de retorno sí es obligatorio. Si no se devuelve nada se usa la palabra reservada **void**
  - Si no hay argumentos no se pone nada entre los paréntesis

3.6 PASO DE PARÁMETROS

- En java los parámetros siempre se pasan por copia
- Los argumentos de los tipos básicos no quedan modificados fuera del método aunque se modifiquen dentro
- En el caso de pasar objetos como parámetro, lo que se copia es una referencia al objeto
  - Si modificamos el objeto dentro del método también se modifica fuera

3.7 VALOR DE SALIDA DE UN MÉTODO

- Se indica con la palabra clave **return** seguida de lo que se quiere devolver

```
public int cuadrado (int x) {
return (x*x); }
```

- Si se devuelve un **void** no es necesario utilizar **return**
- Es conveniente tener un solo **return** por función
- Hay que asegurarse de que, si se devuelve un valor, siempre se puede hacer el return (si no, da un error de compilación)
- Si lo que devolvemos es un objeto:
  - Se actúa igual que en el caso de devolver un valor de un tipo simple

3.8 MÉTODOS ESPECIALES

toString:

- Por defecto, Java convierte cualquier cosa en un String
- Con objetos, si no le decimos cómo hacerlo, el String contiene la dirección del objeto
- La implementación de toString indica cómo hacer la conversión

```
public String toString () {
        return (“(“+c_x+”,”+c_y+”)”+”,”+r);
}
```

equals:

- No se debe usar == para comparar objetos, pues el resultado indica si los dos valores que se comparan son el mismo objeto, no si dos objetos son iguales

```
public boolean equals (Circulo c) {
    return (this.c_x==c.c_x &&
    this.c_y==c.c_y && this.c_r==c.c_r);
}
```

3.8 MÉTODOS ESPECIALES

clone:

- Si a una variable le asignamos otra que contiene un objeto, ambas variables tienen una referencia al mismo objeto
- Para que se asigne una copia del objeto hay que hacerlo a través del método clone

```
public Circulo clone()
{
     return new Circulo(this.c_x, this.c_y, this.c_r);
}
```

3.9 MODIFICADOR STATIC

- En atributos indica que son atributos de clase, es decir, que tienen el mismo valor para todos los objetos de la clase
- Si un objeto modifica el valor, se modifica para todos los objetos
- En métodos indica que son métodos de clase, es decir, que se invocan sobre la clase sin necesidad de crear objetos.
- Sólo pueden manejar atributos static

Ejemplos: método main, métodos de la clase Math

4. HERENCIA

- Tipo especial de relación entre clases
- Es uno de los aspectos que distinguen el Paradigma de Orientación a Objetos frente a otros paradigmas
- Mecanismo que, bien utilizado, facilita la modificabilidad y reutilización de los diseños y el código
- En qué consiste?
  - Existen dos clases, a las que llamaremos padre (superclase o clase base) e hija (subclase o clase derivada)
  - Al igual que las herencias en la vida real, la clase hija pasa a tener lo que tiene la clase padre
    - Atributos
    - Métodos
  - Un objeto de la clase hija es también un objeto de la clase padre
  - En la clase hija se definen las diferencias respecto de la clase padre
- ¿Para qué se usa?
  - Para extender la funcionalidad de la clase padre
  - Para especializar el comportamiento de la clase padre

4. HERENCIA. CONCEPTO

- La herencia modifica el mecanismo de paso de mensajes
- Cuando a un objeto de una clase **C** se le pasa un mensaje **M**, se busca un método **M** en la clase **C**:
  - Si existe un método **M** en la clase **C**, se ejecuta ese método y termina el proceso
  - Si en la clase **C** no hay ningún método **M**, se busca éste en la superclase de **C**
  - Si en la superclase de **C** existe un método **M**, se ejecuta ese método y termina el proceso
  - Si en la superclase de **C** no hay ningún método **M**, se busca en las superclases de la superclase hasta que o bien se encuentra y se ejecuta o no se encuentra en ninguna de las superclases, de forma que el objeto no entiende ese mensaje **M**, dando como resultado el consiguiente error

4. HERENCIA. CONCEPTO

- Ventajas
  - Se ahorra código
  - Permite reutilizar código extendiendo su funcionalidad
- Desventajas
  - Se ahorra código
  - Se introduce una fuerte dependencia en la clase hija respecto a la clase padre
  - Puede dificultar la reutilización
  - Un cambio en la clase padre puede tener efectos imprevistos en las clases hijas
  - Un objeto de una clase hija puede tener un comportamiento inconsistente con lo esperado de un objeto de la clase padre
  - Se establece una jerarquía o clasificación. Si cambia el criterio de clasificación puede acarrear muchas modificaciones

4. HERENCIA. IMPLEMENTACIÓN EN JAVA

- Se indica usando la palabra reservada **extends**

  `class Punto3D extends Punto2D`

- Visibilidad:
  - Los miembros privados de la superclase no son visibles desde la subclase
  - Los miembros públicos de la superclase son visibles y siguen siendo públicos en la subclase
- Se puede acceder a los miembros de la superclase usando la palabra reservada **super**
- Si una clase se declara como final no se puede heredar de ella
- En java, todas las clases heredan implícitamente de la clase **Object**.
- Sólo se puede hacer herencia de implementación (extends) de una clase
- No se crean problemas de referencias circulares o alternativas a un método con la misma declaración

```
class Punto2D {
        private int x,y;
        Punto2D(int x, int y){
                this.x=x;this.y=y;
        }
        public void pintar(){
                …
        }
}

final class Punto3D extends Punto2D {
        private int z;
        Punto3D (int x, int y, int z) {
                super (x,y);
                this.z=z;
        }
        public void pintar() {
                super.pintar();
                …
        }
}
```

4. HERENCIA. CLASES ABSTRACTAS

- En ciertos casos, una clase se diseña directamente para ser extendida por un conjunto de subclases
- En estos casos suele interesar no implementar alguno de sus métodos, pues no tiene ningún significado en la clase base.
- Es necesario declarar tanto la clase como los métodos no implementados como abstractos a través de la palabra reservada **abstract**
- Una clase abstracta es, por tanto, aquella que tiene alguno de sus métodos definido pero no implementado

- Podemos crear una clase Animal a partir de la cual crearemos otras clases que definan animales concretos
- (Casi) todos los animales emiten algún sonido, pero no hay ninguno común para todos los animales
- Cada subclase re implementará el método sound() como le convenga

```
public abstract class Animal {
        private String nombre;
        public abstract void sound();
        public Animal (String nombre) {
                this.nombre=new String(nombre);
        }
        ...
 }
```

4. HERENCIA. INTERFACES

- A veces nos interesa que todos los métodos de una clase abstracta sean abstractos
- Lo usamos para obligar a que todas las subclases reimplementen esos métodos
- Para estos casos, Java proporciona unas clases especiales llamadas interfaces
- Se declaran como **interface**, no como **class**
  Representan el concepto de clase abstracta pura
- Una clase declarada como interfaz no puede tener ningún método implementado

- Para heredar de una interfaz se usa la palabra reservada **implements**

```
public class MiClase implements Serializable
```

- Reglas/Rerstricciones:
  - Una interfaz puede heredar de otra interfaz
  - Una clase (abstracta o no) puede heredar de una interfaz
  - Una interfaz NO puede heredar de una clase
    Es un mecanismo muy usado en Java
  - Se puede hacer herencia de interfaz (implements) de todas las interfaces que se quiera

5. POLIMORFISMO

- Gracias a la herencia, se puede interpretar que un objeto de una subclase es también un objeto de una superclase
- El polimorfismo es un mecanismo que se aprovecha de la herencia (especialmente de interfaz) para manejar indistintamente objetos de las subclases como si fuesen objetos de la clase base, sin preocuparse por la clase en concreto a la que pertenecen
- Interesa utilizarlo cuando un comportamiento varía en función del tipo de algo
- Se declaran atributos, parámetros o variables de la clase base
- Se les asignan objetos de alguna de las subclases

5. POLIMORFISMO

- Estamos seguros de que se pueden usar todos los métodos declarados en la clase base
- Si necesitamos usar métodos de las subclases es necesario hacer un **cast**
- La utilización del cast aumenta la posibilidad de hacer conversiones erróneas, por lo que es mejor evitarlo
- Se puede preguntar por la clase a la que pertenece un objeto:

```
instanceof objeto.getClass().getName()
```

- Es una mala idea. En general, preguntar por la clase de un objeto implica un mal diseño y problemas para modificar el diseño y el código

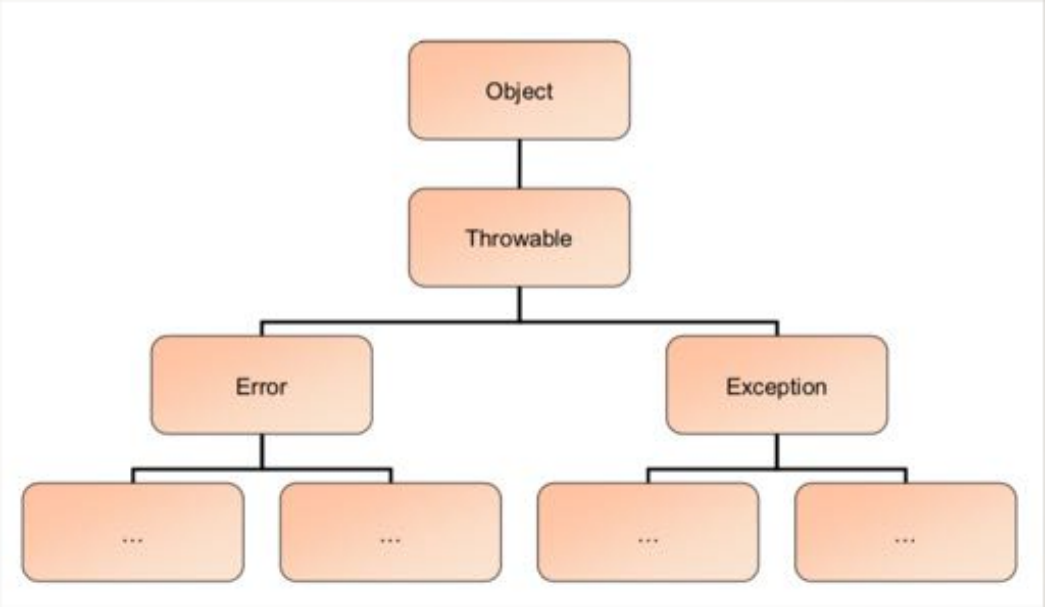
6. EXCEPCIONES

- ¿Qué són?
  - Mecanismo de control de errores en tiempo de ejecución
  - Una forma de hacer que la aplicación continúe la ejecución si se produce un error

  ¿Qué sucede si no se puede abrir un fichero? ¿Qué pasa si un socket se cierra de manera inesperada? ¿Y si dividimos por cero?
- ¿Para qué sirven?
  - A veces se usan para detectar situaciones inusuales en la ejecución
  - Si se controlan todos los posibles errores directamente, el código se puede volver ilegible
  - Con las excepciones se separa el código de un método del código que controla los errores
- Como casi todo lo demás en java, las excepciones son objetos (que se crean cuando ocurre una situación anómala)
- Se ***lanzan*** para que otra parte del código las ***capture*** y las trate

6. EXCECIONES. JERARQUIA

- Las excepciones que se derivan de **Error** suelen estar relacionadas con la maquina virtual y no se espera que se capturen ni se traten
- Las Excepciones derivadas de **Exception** sí que deben ser tratadas, y en algunos casos es obligatorio hacerlo para que el programa compile
- No es necesario tratar las Excepciones que derivan de **RuntimeException**:
  - ArithmeticException
  - IndexOutOfBoundsException
  - NullPointerException
  - SecurityException
  - ...



6. EXCEPCIONES. TRATAMIENTO

- El código que puede generar la excepción debe encerrarse dentro de un bloque **try**

```
try
{
      // Código que puede generar la excepción
}
```

- A continuación, la excepción se captura con un
- bloque **catch**

```
catch (Exception e) {
      // Código para tratar el error
}
```

- El código dentro del try puede generar más de una excepción, y se pueden capturar todas ellas

```
try {
      //Código que puede provocar el error
}
catch(IOException ioe) {
      //Código para tratar la IOException
}
catch(Exception e) {
      //Código para tratar la Exception
}
```

- Esto último se puede hacer gracias a que unas excepciones heredan de otras
- Sólo se puede hacer con excepciones dentro de la misma jerarquía
- Sirve para tratar de manera común varios tipos de excepciones distintos

6. EXCEPCIONES. TRATAMIENTO

- **CUIDADO**: Si un bloque de código lanza varias excepciones y se usan varios catch
  - La excepción se captura en el primer catch que se ajusta a la excepción
  - Los catch deben capturar las excepciones más concretas en primer lugar, y las más generales al final
  - Si no lo hacemos así, hay bloques catch en los que no se entrará nunca

```
try {
        //código que genera
        //excepciones
}
catch(IOException ioe){
        //catch accesible
}
catch (Exception e ){
        //catch accesible
}

try {
        //código que genera
        //excepciones
}
catch(Exception e){
        //catch accesible
}
catch (IOException ioe ){
        //catch NO accesible
}
```

6. EXCEPCIONES. TRATAMIENTO

- A veces, cuando se produce una excepción, la aplicación queda es un estado inestable
- Al tratamiento de una excepción se le puede añadir al final un bloque finally que se ejecuta siempre, se produzcan o no excepciones
- Se puede usar para cerrar ficheros, liberar recursos, etc.

```
try {
        //código que genera excepciones
}
catch(IOException ioe) {
        //tratamiento de la excepción
}
 finally {
        //código que se ejecuta siempre
}
```

- Si una excepción no se captura se propaga hacia el método llamante, para que éste la trate
- Si no la trata, se sigue propagando hasta llegar al main
- Si en el main tampoco se trata, se aborta la ejecución del programa

6. EXCEPCIONES. LANZAMIENTO Y CREACIÓN

- **LANZAMIENTO**
  - Esto se hace utilizando la palabra reservada **throws** en la cabecera del método

    ```
    public void miMetodo() throws ArithmeticException
    ```

  - Posteriormente, en el código, se puede lanzar una excepción usando la palabra reservada **throw**

```
try{
        …
}catch (IOException ioe) {
        throw ioe;
}catch (Exception e){
        throw new NullPointerException();
}
```

- **CREACIÓN**
  - Se pueden crear nuevas excepciones creando una nueva clase que herede de **Exception**
  - Se suele añadir un constructor por defecto y otro que acepta un String
  - Llamar al constructor de Exception desde el constructor de la nueva clase
  - Si hace falta, añadir atributos, otros constructores y otros métodos

7. OTRAS CARACTERÍSTICAS OO. MODULOS

- En Java hay tres tipos de módulos:
  - Métodos
  - Clases
  - Paquetes
- El API (Application Programming Interface) de Java ofrece varios módulos :

- FICHEROS
  - El código de una clase pública se encuentra en un fichero .java
  - El nombre del fichero coincidirá con el nombre de la clase
  - **Excepción**: En un fichero podría haber varias clases, si sólo una de ellas es public y el resto clases auxiliares

7. OTRAS CARACTERÍSTICAS OO.

- PAQUETES
  - Los paquetes contienen a un conjunto de clases relacionadas
  - Los paquetes ayudan a manejar la complejidad de las aplicaciones
  - Facilitan la reutilización del SW permitiendo a los programas el importar clases de otros paquetes
  - Puede haber más de una clase en el mismo fichero fuente … pero sólo una puede ser pública y debe coincidir en nombre con el fichero
  - Lo habitual suele ser tener una clase por cada fichero fuente
  - El paquete al que pertenece una clase se indica al comienzo del fichero
  - Una clase sólo puede pertenecer a un paquete
  - Se pueden organizar los paquetes de forma jerárquica

    `package ejemplo.objetos;`
  - Para usar una clase que está en distinto paquete: Se puede importar la clase entera
  - Se pueden realizar llamadas utilizando `nombrepaquete.resto`

7. OTRAS CARACTERÍSTICAS OO.

- IMPORTAR CLASES
  - Se puede importar un conjunto de clases
  - O una clase concreta

```
import ejemplo.*;
import ejemplo.Concreto;
```

- VISIBILIDAD, CONTROL DE ACCESO
  - Puede (y suele) haber distintos niveles de visibilidad para los miembros (atributos y métodos) de una clase.
  - Existen cuatro tipos de modificadores:
    - **public**: se puede acceder desde cualquier lugar
    - **private**: sólo se puede acceder desde la propia clase
    - **protected**: sólo se puede acceder desde la propia clase o desde una clase que herede de ella
    - Por defecto (package): Se puede acceder desde las clases pertenecientes al mismo paquete

7. OTRAS CARACTERÍSTICAS OO.

- USO DE LIBRERÍAS EXTERNAS

  Igual que en un proyecto java está disponible toda la estructura de ficheros de clases de la aplicación, podemos añadir librerías de terceros (previamente almacenadas como jar).

  Se pueden añadir mediante el uso del IDE (en eclipse "propiedades del proyecto" → java build path → libraries o añadiendo la ruta al archivo jar cuando se usa el comando javac

7. OTRAS CARACTERÍSTICAS OO.

- GENERICIDAD y COLECCIONES EN JAVA
  - Facilidad de un lenguaje de programación para definir clases parametrizadas con tipos de datos.
  - Resultan de utilidad para la implementación de tipos de datos **contenedores** como las **colecciones**.
  - La genericidad sólo tiene sentido en lenguajes con comprobación estática de tipos, como Java.
  - La genericidad permite escribir código reutilizable

```
public class Contenedor<T> {
        private T contenido;
        public void setContenido(T contenido) {
                        this.contenido = contenido;
        }
        public T getContenido() {
                return contenido;
        }
 }
// Parametrización de un contenedor
Contenedor contenedor = new Contenedor<String>();
contenedor.setContenido("hola");
```

- COLECCIONES POR DEFECTO EN JAVA
  - Conjuntos → Set: HashSet, TreeSet,LinkedHashSet
  - Listas → List: ArrayList,LinkedList,Vector
  - Mapas → Map: HashMap,TreeMap,LinkedHashMap

Importante: Clase **Iterator**

Descripción ampliada:
http://www.jtech.ua.es/j2ee/publico/lja-2012-13/sesion02-apuntes.html
Direrencias entre list,set,map:
http://vayajava.blogspot.com.es/2008/05/diferencias-entre-las-colecciones-list.html

> ***TIP***: *En este caso y en todo aquello que no conozcamos como funciona exactamente, una buena forma de comenzar es buscar en internet por <Concepto> javadoc, para obtener la descripción de su API*