Tema 7:
P.O.O. Utilización avanzada de clases

1. Sobrecarga de métodos
2. Herencia. Superclase y subclases
3. Sobreescritura de métodos.
4. Clases y métodos abstractos
5. Polimorfismo
6. Paquetes
7. Enumerados
8. Interfaces

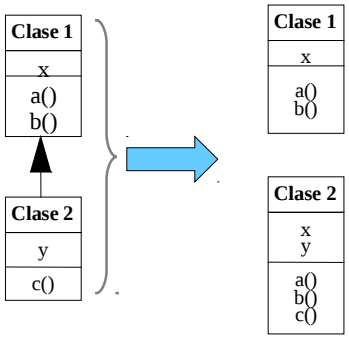
1. Sobrecarga de métodos

- Un método sobrecargado es un método que tiene el mismo nombre que otro pero con diferentes argumentos.
- Reglas
  - Siempre debe ser diferente la lista de argumentos.
  - Puede cambiar el tipo de retorno
  - Puede cambiar el modificar de acceso
  - Un método puede ser sobrecargado en la misma clase o en una subclase (lo veremos en la herencia)
  - Los constructores también se pueden sobrecargar (ver en "EjemploSobrecarga.java")

2. Herencia

- La herencia es un mecanismo que permite crear clases a partir de otras existentes:
  - Heredando y posiblemente añadiendo atributos
  - Heredando y posiblemente añadiendo y/o modificando métodos
- Las clases pueden heredar características de otras clases, lo que permite aumentar la reutilización del software:
  - La clase de la que se hereda se denomina superclase o clase padre.
  - La clase que hereda se denomina subclase o clase hija.
- La herencia puede aplicarse muchas veces lo que permite crear una jerarquía de clases.

Ejemplo. Clase 2 hereda de Clase 1

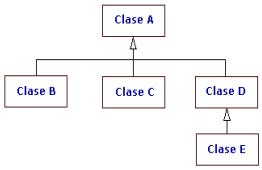


Ejemplo

- En un centro de enseñanza tenemos la clase Persona
  - Atributos de persona: nombre, dni, direccion
  - Metodos: setNombre, setDireccion, obtenerDatosPorDni, ..
- Creamos la clase Alumno que hereda de Persona. Además de los atributos y métodos de Persona la clase Alumno tiene
  - Atributos: curso, notas (array)
  - Metodos: ponerNotaEnAsignatura, consultarCurso,...
- Creamos la clase Profesor que hereda de Persona. Además de los atributos y métodos de Persona la clase Profesor tiene
  - Atributos: especialidad, horario, antiguedad
  - Métodos: getEspecialidad, mostrarHorario, ..

Jerarquía en la herencia

- A es la superclase de B, C y D.
- D es la superclase de E.
- B, C y D son subclases de A.
- E es una subclase de D



Herencia en Java

En Java, una clase sólo puede extender una superclase (herencia simple). Se utiliza la cláusula ***extends***:

```
public class nomSubclase extends nomSuperclase{
        // codigo de la subclase
      }

public class Clase2 extends Clase1 {
      // código de la Clase 2
}
```

NOTA: Para implementar la herencia múltiple en JAVA se usan las interfaces

Ejemplo herencia Java

```
public class Vehiculo {
    private String  numSerie;
    private String  color;
    // metodos get y set
}
public class Coche extends Vehiculo
{
    private int npuertas;
    private int numAirbags;
    public void realizarItv(){
    }
}
public class Camion extends Vehiculo
{
    private int pma; // peso maximo autorizado

    public void paradaRegulada (){
    }
}
```

Visibilidad

Modificadores de visibilidad de los atributos

- **public:** accesible desde cualquier clase.
- **protected:** accesible desde la propia clase y de sus subclases. Ojo: también desde la clase del mismo paquete.
- **private:** sólo accesible en la propia clase.
- Sin ningún modificador significa que es accesible desde la propia clase y desde cualquier clase del mismo paquete (se suele llamar visibilidad friendly o visibilidad de paquete).

Visibilidad

Visibilidad	Clase	Paquete	SubClase	Todos
Public	SI	SI	SI	SI
Private	SI	NO	NO	NO
Protected	SI	SI	SI	NO
Friendly	SI	SI	NO	NO

this y super

- **this** referencia al objeto actual en que se está ejecutando el código:
  - **this**: permite referenciar a los atributos y métodos del objeto actual
  - **this()**: hace referencia al constructor del objeto actual.
- **super** referencia al objeto de la superclase del objeto actual en que se está ejecutando el código
  - **super**: permite referenciar a los atributos y métodos del objeto de la superclase.
  - **super()**: hace referencia al constructor del objeto de la superclase.

this y super

En ocasiones es imprescindible el uso de this para resolver ambigüedades

```
public class Alumno {
   private int edad;
   private String nombre;

   public Alumno (String nombre, int edad){
         this.edad= edad;
         this.nombre= nombre;
      }
   }
```

this y super

- En ocasiones es imprescindible el uso de super para resolver ambigüedades

```
public class Persona{
    private String nombre;
    public String toString (){
        return "Nombre" + nombre;
    }
}
public class Empleado extends Persona{
    private int sueldo;
    public String toString (){
        return super.imprime ()  + "Sueldo"  + sueldo;
        // sin super generaría que se llamara a si mismo de
        // forma infinita (recursividad infinita)
    }
}
```

Constructores y Herencia

- Las subclases deben definir su propio constructor
- Normalmente será necesario inicializar los atributos de la superclase. Para ello se llama a su constructor desde el de la subclase

```
/* Constructor de una subclase */
public Subclase(parámetros...) {
   // invoca el constructor de la superclase
   super(parámetros para la superclase);
   // inicializar sus atributos
   ...
}
```

- La llamada a "super" debe ser la primera instrucción del constructor de la subclase

Constructores y herencia

- Si desde un constructor de una subclase no se llama expresamente al de la superclase el compilador añade la llamada al constructor sin parámetros
- En el caso de que la superclase no tenga un constructor sin parámetros se produciría un error de compilación.

Constructores y Herencia

- Las subclases deben definir su propio constructor
- Normalmente será necesario inicializar los atributos de la superclase. Para ello se llama a su constructor desde el de la subclase

```
/* Constructor de una subclase */
public Subclase(parámetros...) {
   // invoca el constructor de la superclase
   super(parámetros para la superclase);
   // inicializar sus atributos
   ...
}
```

- La llamada a "super" debe ser la primera instrucción del constructor de la subclase

Sobreescritura de métodos

- Cada vez que se tiene una clase que hereda un método de una superclase, se tiene la oportunidad de sobreescribir el método (a menos que dicho método esté marcado como *final*).
- El beneficio de sobreescribir un método heredado es la posibilidad de definir un comportamiento específico para los objetos de la subclase
- Ya lo hemos hecho cada vez que programamos el toString o el equals. Estamos sobreescribiendo el método de la clase Object.

3. Sobreescritura de métodos

Las reglas para la sobreescritura de métodos son:

- La lista de argumentos del método debe ser la misma.
- El tipo de retorno debe de ser el mismo o un subtipo del tipo de retorno declarado originalmente.
- El nivel de acceso no debe de ser más restrictivo pero puede puede ser menos restrictivo (De más a menos restrictivo el orden es private, friendly, protected, public)
- El método sobrescrito puede lanzar menos excepciones que el método base. No puede lanzar excepciones nuevas.
- No se puede sobreescibir un método marcado como **final** ni marcado como **static**.

Sobreescritura de métodos

Ver ejemplo completo en “EjemploSobreescritura.java”

```
public class Rectangulo {
   private double lado1;
   private double lado2;
   // .....
   public void dibujar (){
        System.out.println("Dibujando rectángulo con lados " + lado1 + " " + lado2);
   }
}
public class Cuadrado extends Rectangulo{
   public void dibujar () // Este método está sobreeescrito
   {
        System.out.println("Dibujando cuadrado con lado " +
   super.get_lado1());
   }
}
```

Clases y métodos final

- Una clase final es aquella de la que no se podrá heredar. Es decir es una clase que no puede tener ninguna subclase.
- La utilidad de poner una clase como final es asegurarse que su comportamiento siempre será ese. Ninguna clase podrá heredar de ella y modificar su comportamiento, por lo que proporciona seguridad en ese sentido. Por ejemplo, la clase String es final.
- Si un método es final no se podrá sobreescribir. Del este modo si queremos que ninguna clase que herede este método cambie su comportamiento lo declararemos como final.

4. Clases Abstractas

- Una clase abstracta es una clase de la que no se pueden crear objetos. Su utilidad es permitir que otras clases deriven de ella
- Un método abstracto es un método declarado en una clase para el cual esa clase no proporciona la implementación (el código).
- Una clase abstracta puede contener métodos no abstractos y método abstractos. No es obligatorio que tenga un método abstracto.
- Una clase derivada de una clase abstracta debe implementar los métodos abstractos (escribir el código) o bien volverlos a declarar como abstractos, con lo que ella misma se convierte también en clase abstracta.

Clases Abstractas

- Se pueden crear referencias a clases abstractas como a cualquier otra clase
  `Figura f;`
- Sin embargo una clase abstracta no se puede instanciar, es decir, no se pueden crear objetos de una clase abstracta. Error:
  `Figura f = new Figura(); // ERROR`
- Sin embargo utilizando el up-casting (convesión hacia arriba) es posible hacer:
  ```
  Figura f= new Rectangulo(. . .);
  f.area();
  ```
  La invocación al método `area` se resolverá en tiempo de ejecución y la JVM llamará al método de la clase adecuada. En nuestro ejemplo se llamará al método `area` de la clase `Rectangulo`.

5. Polimorfismo

- ***Un objeto solamente es de una clase*** (la que se le asigna cuando se construye ese objeto, es decir cuando se hace el new)
- La referencia a un objeto es ***polimórfica*** (puede tomar varias formas) porque puede referirse a objetos de diferentes clases. Para que esto sea posible debe haber una relación de ***herencia*** entre esas clase

```
Figura figura;
 // Creamos un objeto de tipo Rectangulo, aunque el tipo de la referencia
 // es Figura.
 figura= new Rectangulo(4,4, 2, 1);

 //Ojo: al reves no podría hacerse porque un Rectángulo es una Figura pero
 // NO al contrario
 //Circulo ciruculo= new Figura (....)  ERROR!!!

 // Creamos un objeto de tipo Circulo, aunque el tipo de la referencia
 // es Figura
 figura=new Circulo(9,7, 6);
```

- La combinación de la ***herencia*** y el ***enlace dinámico*** es lo que se conoce como ***polimorfismo***

Enlace dinámico

Cuando tenemos clases que heredan de otras, Java permite decidir a que método llamar en tiempo de ejecución, esto se conoce como enlace dinámico

El método se selecciona en base a tipo de objeto, no al tipo de la referencia

```
public class EjemploEnlaceDinamico {
public static void main(String[] args) {
     Figura f;
     // Creamos un objeto de tipo Rectangulo, aunque el tipo de la referencia
     // es Figura
     f= new Rectangulo(4,4, 2, 1);

     // En tiempo de ejecución se decide a que método llamar, en este caso
     // el método área de Rectángulo
     System.out.println("La figura (ahora es un rectángulo) tiene área "  + f.area());

     // Creamos un objeto de tipo Circulo, aunque el tipo de la referencia
     // es Figura
     f=new Circulo(9,7, 6);

     // En tiempo de ejecución se decide a que método llamar, en este caso
     // el método area de Circulo
     System.out.println("La figura (ahora es un círculo) tiene área "  + f.area());
    }
}
```

Polimorfismo y Enlace dinámico

- Ejemplo: Creamos un array de objeto Figura

```
Figura[] arrayFiguras=new Figura[3];
arrayFiguras[0]=new Rectangulo(0,0, 5, 7);
arraFiguras[1]=new Circulo(0,0, 8);
arrayFiguras[2]=new Circulo(7, 0, 3);
```

- La sentencia

```
arrayFiguras[i].area();
```

- ¿A qué método *area* llamará?. La respuesta será, según sea el índice *i*. Por tanto la decisión sobre qué función *area* se va a llamar se retrasa hasta el tiempo de ejecución.
- Ver ejemplo completo en "EjemploPolimorfismo.java"

Operador instanceof

- El operador instanceof sirve para saber el tipo de un objeto. Devuelve true o false
- Es muy útil en casos de herencia
- La sintaxis es **objeto instanceOf Clase**
- Ejemplo: una clase ClaseB que hereda de ClaseA

  ClaseA **a** = new ClaseA()
- ¿Que devolverá **a instanceOf ClaseA**? true
- ¿Qué devolverá **a instanceOf ClaseB**? false

Operador instanceof

- Un objeto hijo sí es instancia del objeto padre
- Ejemplo: una clase ClaseB que hereda de ClaseA

  ClaseB **b** = new ClaseB()
- ¿Que devolverá **b instanceOf ClaseB**? true
- ¿Qué devolverá **b instanceOf ClaseA**? true

Conversión entre objetos

- Utilizar instaceOf para verificar el tipo de objeto
- La conversión hacia clases superiores en la jerarquía se hace implicitamente (mediante la asignación)
- La conversión hacia abajo ha de indicarse con conversión explícita (Casting). El compilador la comprobará en tiempo de ejecución y si hay error generará la excepción ClassCastException

6. Paquetes

- Las clases se agrupan en paquetes de acuerdo con su funcionalidad. Son independientes de la jerarquía de clases
- La declaración de paquetes debe ser la primera instrucción válida de un clase

```
package paquete;
```

- Uso de las clases definidas en un paquete:

```
import paquete.clase;
import paquete.*;
```

- Los paquetes relacionados se pueden agrupar en un único paquete. Un ejemplo es el paquete **java** que contiene a otros paquetes como **lang**, **util**, **io**, etc.
- La convención para los nombres de paquetes es utilizar el nombre del dominio seguido del nombre de la organización. Por ejemplo

com.sun.eng

7. Enumerados

- Un tipo enumerado en Java es un tipo restringido a un conjunto de valores

```
public enum TipoDemarcacion{
PORTERO, DEFENSA, CENTROCAMPISTA, DELANTERO
}

public enum TipoEstadoCivil{
SOLTERO,CASADO, VIUDO,DIVORCIADO
}
```

- Pueden crearse como una clase independiente o dentro de otra.
- Los valores por convención se ponen en mayúscula

7. Enumerados

- Crear un atributo de tipo enumerado

```
private TipoEstadoCivil estadoCivil;
```

- Asignar un valor a un enumerado

```
estadoCivil=TipoEstadoCivil.SOLTERO;
```

- Mostrar el valor. El enumerado tiene redefinido su toString así que podemos ver el valor de esta forma

```
System.out.println("El estado civil es " +
  estadoCivil)
```

7. Enumerados

- Método **`valueOf (String)`**: Método estático del enumerado. Sirve para pasar del String al objeto de tipo enumerado.

Genera la excepción **`IllegalArgumentException`** si el String no coincide con ningún valor del enumerado

- Metodo **`values()`**: Método estático del enumerado. Devuelve un array de String con los valores del enumerado

```
TEstadoCivil estadoCivil;
String cadena;
System.out.println("Introduce estado civil:" +
  Arrays.toString(TEstadoCivil.values()) );
cadena=teclado.nextLine().toUpperCase();
estadoCivil= TEstadoCivil.valueOf(cadena);
```

8. Interfaces

- **Interfaz**: un conjunto de constantes y métodos públicos sin cuerpo (es decir sin código,como si fuesen métodos abstractos)
- Se encarga de establecer unas líneas generales sobre los comportamientos (métodos) que deberían tener los objetos de toda clase que implemente esa interfaz, es decir, que no indican lo que el objeto es sino acciones (**capacidades**) que el objeto debería ser capaz de realizar
- Muchas interfaces en Java termina con sufijos del tipo "-able", "-or", "-ente" que significan algo así como capacidad o **habilidad para hacer** o ser receptores de algo (configurable, serializable, modificable, clonable, ejecutable..)
- Ejemplo: Interfaz Arrancable con un método arrancar() que se implementada por algunos tipos de vehículos (Coche, Moto) pero no por otros (Bicicleta)

8. Interfaces

- **Difencia entre clase abstracta e interfaz:** Una clase abstracta puede incluir métodos implementados y no implementados o abstractos, atributos constantes y otros no constantes, sin embargo, una interfaz solo puede incluir métodos no implementados y declaración de constantes.
- Los atributos de una interfaz son siempre **public**, **static** y **final**
- Los métodos de una interfaz son siempre **public** y **abstract**
- Una clase puede heredar (extend) de una sola clase pero **puede implementar varias interfaces**.
- Ejemplo: Podría existir otro interfaz llamada MedibleConsumo con un metódo medirConsumo(). La clase Coche puede implementar la interfaz Arrancable y MedibleConsumo.
- Las interfaces en JAVA pueden usarse para implementar la herencia múltiple

8. Interfaces

[modificadores] **interface** nombre [cláusulaExtends]
{
[cuerpo]
}
modificadores:

- Sobre visibilidad: **public** (interfaz pública) o nada (interfaz no pública); es decir, igual que las clases
- **abstract** (aunque no hace falta ponerlo, ya que cualquier interfaz es **abstract** por defecto)

cláusulaExtends: **extends** interfaz1, ..., interfazN

cuerpo: conjunto de constantes y de prototipos de métodos (por tanto, sin cuerpo y acabados en punto y coma)

8. Interfaces

- Una clase implementa una interfaz si contiene una implementación para cada uno de los métodos declarados en la interfaz
- También se permite que la clase sólo implemente algunos de los métodos de la interfaz; en este caso, quedarían métodos abstractos sin cuerpo, por lo que la clase se convertiría en abstracta
- Una clase puede implementar varias interfaces, con la condición de que contenga un método público con código concreto por cada una de las operaciones de cada interfaz
- Cuando definimos una clase, indicamos la lista de interfaces que la clase implementa

8. Interfaces

[modificadores] **class** nombre [cláusulaExtends] [cláusulaImplements]
{
  [cuerpo]
}

cláusulaImplements:
    implements interfaz1, ..., interfazN

- Las interfaces se diseñan dentro de los paquetes, igual que las clases, y tienen las mismas visibilidades (pública y de paquete)
- En Java existe una jerarquía de interfaces ya definidas como en el paquete java.lang: Cloneable, Comparable, Runnable, Serializable
- Entre las interfaces también puede existir relación de herencia