1º CFGS Desarrollo De Aplicaciones Multiplataforma

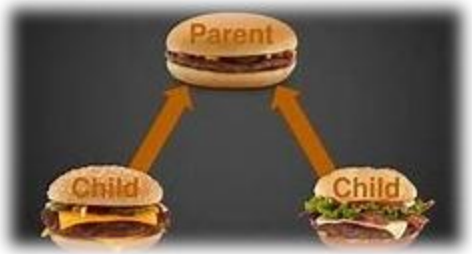
UD 6. Programación orientada a objetos. Avanzado.

Módulo: Programación

Centro de Enseñanza
Gregorio Fernández

Macarena Cuenca Carbajo

Herencia



- La **herencia** es una característica de la programación orientada a objetos que permite crear una clase (**clase hija/derivada**) partiendo de otra ya creada anteriormente (**clase padre/base**).
- La nueva clase hereda el **estado** y **comportamiento** de la clase base.
- Con ello se consigue:
  - La **reutilización** del código.
  - Definir una **jerarquía** de clases.
  - Eliminar código **redundante**.
- La herencia es una poderosa herramienta que permite producir software adaptable y reutilizable.

Herencia

- Hay dos tipos de herencia:
  - **Herencia simple:** sólo se puede heredar de una clase base.
  - **Herencia múltiple**: se puede heredar de más de una clase base.
- Java sólo permite la herencia simple.

Clases derivadas

- Ambas clases tendrán en común un **comportamiento** (métodos) y un **estado** (datos).
- Además, la clase **Programador** tendrá sus características propias.

- Si creamos la clase derivada de forma independiente duplicaríamos código ya existente en la clase base.
- Gracias a la herencia esto no ocurre, y hacemos que la clase Programador sea una extensión de la clase Empleado.

Clases derivadas
I - Declaración

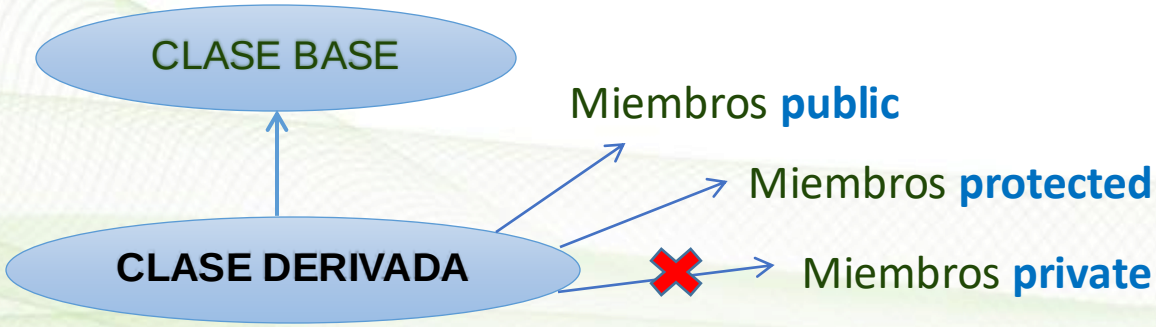
- Sintaxis:

```
class NombreClaseDerivada extends NombreClaseBase
```

- La palabra reservada **extends** hace que todos los miembros **no private** de la **clase base** sean heredados en la **clase derivada**.

Clases derivadas
II – Accesibilidad

- Una **clase derivada** no puede acceder a los miembros **privados** de su **clase base**.
- Pero la **clase base** puede declarar determinados miembros protegidos (**protected**) para ocultar detalles de su implementación a clases no derivadas, y sí permitir que dichos miembros sean visibles desde sus **clases derivadas**.



Macarena Cuenca Carbajo
Centro de Enseñanza
Gregorio Fernández

Constructores

- Sintaxis del constructor de una **clase derivada**:

```
public ClaseDerivada ([lista parámetros]) {
        super([lista parámetros clase base]);
        //cuerpo constructor
}
```

- Los constructores de las **clases base** son invocados antes que los constructores de las **clases derivadas**.
- Se crea un objeto de la **clase base** antes de que el constructor de la **clase derivada** realice su tarea.

Constructores

- La primera línea del constructor de la clase derivada debe ser la llamada a un constructor de la **clase base**.
- Esta llamada se realiza con **super**.
- Si no se indica explícitamente, se asume que se está llamando al constructor por defecto (sin argumentos) de la **clase base**. Si en ella no se ha definido un constructor sin argumentos, entonces dará un error de compilación.

Constructores

- Ejemplo:

```
public class Persona {

  protected String nombre;

  public Persona(String nom){
      this.nombre=nom;
  }
}

public class Joven extends Persona {

  private int edad;

  public Joven(String nom, int edad) {
      super(nom);
      this.edad=edad;
  }
}
```


Macarena Cuenca Carbajo

Centro de Enseñanza
Gregorio Fernández

Sobrescritura de métodos

Overriding

- En una clase derivada es posible **redefinir** métodos heredados.
- Lo que se está haciendo es **sobrescribir** esos métodos.
- Consiguiendo **“ocultar”** los correspondientes métodos de la clase base.
- Si se desea invocar desde la clase derivada al método de la clase base que ha sido redefinido (“ocultado”), se puede realizar de la siguiente forma:

```
super.metodo([lista de parámetros actuales])
```

- La **sobreescritura** de métodos tiene como finalidad añadir funcionalidad a la heredada por defecto.

Sobrescritura de métodos

- Para que un método de la clase derivada sobrescriba a un método heredado, deberá cumplir lo siguiente:
  - Tener el mismo **nombre**.
  - El **tipo de retorno** debe ser el mismo.
  - Tener el mismo nº y tipo de **parámetros**.

Sobrescritura de métodos

- Ejemplo:

```
public class Base {
    public void metodo1(int n) {...}
    public void metodo2(String s, int n) {...}
}

public class Derivada extends Base {
    public void metodo1(int n) {...}
    public void metodo2(String s) {...}
}
```

- El metodo1 de la clase Derivada **sobrescribe** al método1 de la clase Base, con lo cual, lo reemplaza.

Sobrescritura de métodos

- Analicemos el siguiente código:

```
Base b=new Base();
Derivada d=new Derivada();
b.metodo1(8);
d.metodo1(3);
```

b.metodo1(8); ← Llamada al metodo1 de la clase **Base**

d.metodo1(3); ← Llamada al metodo1 de la clase **Derivada** (el método heredado es reemplazado)

Sobrecarga de métodos

Overloading

- La **sobrecarga**, es la implementación del mismo método varias veces, con ligeras diferencias adaptadas a las distintas necesidades de dicho método.
- **Sobrecargar** un método consiste en definir un método con el mismo nombre que otro existente, pero con distinto número de argumentos o distintos tipos de argumentos.
- El **tipo de retorno** no es tenido en cuenta en la sobrecarga, es decir, los métodos sobrecargados pueden cambiar el tipo de retorno.
- El concepto es el mismo que la sobrecarga de constructores.
- La sobrecarga de métodos también se puede realizar en **clases derivadas**, **sobrecargando métodos heredados**.

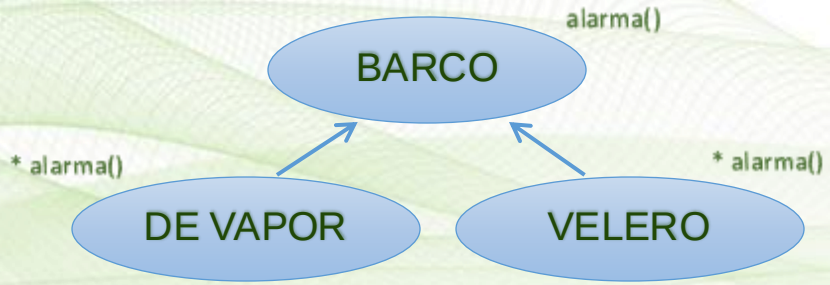
Sobrecarga de métodos

- Si volvemos al ejemplo anterior, el **método2** de la clase Derivada sobrecarga al **método2** de la clase Base.
- Si en la clase **Derivada** añado el siguiente código, ¿qué crees que ocurrirá?

```
public String metodo2(String s, int n) {...}
```

Conversión entre objetos derivados y objetos base

- Para convertir un objeto en otro debe haber entre ambos una relación de herencia.
- Como os objetos de **clases derivadas son también** objetos de su **clase base**, Java realiza el **casting** automáticamente, convirtiendo una referencia a un **objeto derivado** a una referencia a su **clase base**.
- Ejemplo:




```
Barco b; //ref. a la clase base
Velelo v=new Velero();
DeVapor va=new DeVapor();

b=v;
b=va;
```

Conversión automática

Conversión entre objetos derivados y objetos base

- La conversión inversa no es posible sin realizar un **casting**, ya que todo **Barco** no es un **Velero** ni todo **Barco** es un barco **DeVapor**.

```
v=(Velero)b;
va=(DeVapor)b;
```

- Si hacemos la siguiente llamada:

```
b.alarma();
```

- Se ejecutará el método **alarma()** correspondiente al objeto **derivado** asignado a la ref. y no el de la **clase base Barco**.

La **llamada desde una referencia a la clase base** que tiene asignado un objeto derivado, **a un método redefinido** en la **clase derivada**, provoca la ejecución del método de la **clase derivada**.

Conversión entre objetos derivados y objetos base

- De la misma forma, si en un método un parámetro formal es de tipo objeto, el parámetro actual puede ser del mismo tipo o de un tipo inferior.
- Siguiendo con el ejemplo de los barcos, si tenemos el siguiente método:

```
public void metodo(Barco b){...}
```

- Se admiten las siguientes llamadas:

```
metodo(v);
metodo(va);
```

Conversión entre objetos derivados y objetos base

- Dentro del método, para saber de qué tipo es el objeto que se ha recibido, podemos usar el operador **instanceof.**
- Para poder usar los miembros del objeto que se envía, habría que hacer el **casting** correspondiente:

```
public void metodo(Barco b){
  if(b instanceof Velero){
     Velero v=(Velero)b;
     ...
  }else if(b instanceof DeVapor){
     DeVapor va=(DeVapor va)b;
     ...
  }
}
```

- Este tipo de conversiones son esenciales para implementar **métodos polimórficos**.

Ligadura dinámica

- **Ligadura:** conexión entre la llamada a un método y el código que se ejecuta.
- Ésta puede ser de dos tipos en función del momento en que se realiza esta conexión:
  - **Estática :** si se realiza en **tiempo de compilación**.
  - **Dinámica:** si se realiza en **tiempo de ejecución**.
- En la **ligadura estática**, el compilador y enlazador definen directamente la posición del código que se ha de ejecutar en la llamada al método.
- En la **ligadura dinámica**, el código a ejecutarse no se puede determinar hasta el momento de la ejecución.

Ligadura dinámica

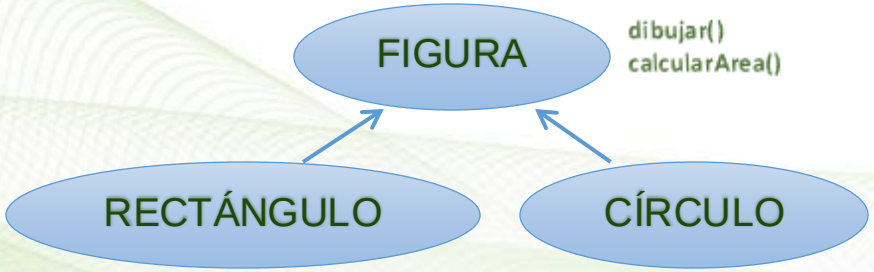
- En Java, la ligadura por defecto es **estática**, mientras que la ligadura **dinámica** tiene lugar en los siguientes casos:
  - → **Redefinición de métodos** en clases derivadas.
  - → **Definición de métodos abstractos**.
- En el ejemplo anterior, la llamada:

```
b.alarma();
```

- Se resuelve tiempo de ejecución, ya que aunque la referencia es de tipo **Barco**, apunta a un objeto de tipo **Velero** o **DeVapor**, y el método de estas clases es el que se va a ejecutar.
- Por tanto, hay **ligadura dinámica**.

Métodos abstractos

- Un **método abstracto** es aquel que es declarado como tal en la **clase base** con la palabra reservada **abstract** y será **definido** en una **clase derivada**.
- El uso de **métodos abstractos** está enfocado a la implementación del **polimorfismo**.
- Analicemos la siguiente jerarquía de clases:



Macarena Cuenca Carbajo
Centro de Enseñanza
Gregorio Fernández

Métodos abstractos

- La **clase base Figura** representa figuras geométricas en el plano.
- Contiene miembros comunes a todas las figuras, como por ejemplo, los métodos **dibujar()** y **calcularArea().**
- Sin embargo, en este punto de la jerarquía no tenemos información suficiente para proporcionar un código genérico para estos métodos.
- Con lo cual, se trata de **métodos abstractos**.
- Serán definidos en cada **clase derivada**, en las que ya si se dispone de información suficiente para implementar su código.

```
public abstract String dibujar();

public abstract double calcularArea();
```

Métodos abstractos

- La clase **Figura** es una **clase abstracta** ya que contiene **métodos abstractos**.
- Una clase con **uno o más métodos abstractos** es una **clase abstracta** y como tal hay que declararla con la palabra clave **abstract**.

```
abstract public class Figura
```

- Ahora, cada clase derivada de **Figura** debe definir sus propias versiones de los métodos abstractos **dibujar()** y **calcularArea()**.
- Si una clase derivada **NO** redefine algún **método abstracto** de su clase base, entonces se convierte en **clase abstracta** ya que hereda el método como **abstracto**.

Centro de Enseñanza
Gregorio Fernández

Clases abstractas

- La utilidad de una clase abstracta es utilizarla para fijar el **comportamiento mínimo** que tiene que tener una clase.
- En Java, **abstract** es sinónimo de genérico.
- Las **clases abstractas** son clases que han sido pensadas para ser genéricas.
- Crearemos entonces **clases abstractas** para representar conceptos generales, características comunes de un tipo de objetos.
- Esto quiere decir que no va a haber objetos de esas clases puesto que no tiene sentido.

Clases abstractas

- Propiedades de una clase abstracta:
  - ➜ Se declara con la palabra reservada **abstract**.
  - ➜ Una clase con **al menos un método abstracto** es una **clase abstracta**.
  - ➜ Una **clase derivada** que **no redefine un método abstracto** heredado es también clase **abstracta**.
  - ➜ Las **clases abstractas** pueden tener **variables de instancia** y **métodos no abstractos**.
  - ➜ **No se pueden crear objetos** de **clases abstractas**, aunque si referencias.

Clases abstractas

- Por ejemplo, podemos definir la **clase abstracta Persona** de la siguiente forma:

```
abstract public class Persona {
    private String nombre;
    private String apellidos;

    public void identificacion(String a, String b) {
            this.nombre=a;
            this.apellidos=b;
    }
}
```

Clases abstractas

- Como vemos, una **clase abstracta** normalmente tiene **métodos abstractos** pero no es condición indispensable.
- Puede disponer de variables y métodos de instancia. Entonces, ¿por qué se declara como **abstracta**?
- La única razón, es definir la clase como modelo que deben de seguir un tipo de objetos e impedir que se puedan crear objetos de esa clase ya que se trata de una **abstracción**.
- Así en el ejemplo, en un contexto de empresa no tiene sentido crear un objeto **Persona** y sí objetos **Empleado**.

Ligadura dinámica con métodos abstractos

- Siguiendo con el ejemplo de las figuras, analicemos las siguientes líneas de código:

```
Rectangulo r = new Rectangulo(2,5);
Circulo c=new Circulo(10);
r.dibujar();
c.dibular();
```

Ligadura estática

- La llamada al método **dibujar()** se puede resolver en **tiempo de compilación**, es decir, el compilador puede determinar que se desea llamar al método **dibujar()** de la clase **Rectangulo** y al método **dibujar()** de la clase **Circulo**.

Ligadura dinámica con métodos abstractos

- En estos casos:

```
Figura f;
Rectangulo r = new Rectangulo(2,5);
Circulo c=new Circulo(10);
f=r;
f.dibujar();
f=c;
f.dibular();
```

Ligadura dinámica

- El compilador no puede determinar los métodos a los que se llamará, ya que depende del objeto asignado a la referencia de tipo **Figura**.
- Únicamente la ejecución del programa lo resolverá. Por eso, en este caso tenemos **ligadura dinámica**.
- No se pueden crear objetos de **clases abstractas**, pero como vemos en el ejemplo, si se puede declarar **referencias** a **clases abstractas** y posteriormente asignarlas **objetos derivados**, con el objetivo de conseguir **ligadura dinámica** en la ejecución de métodos declarados como **abstractos**.

Polimorfismo

- Una de las propiedades más importantes de la POO.
- Es la capacidad que tiene un mismo método de responder de forma distinta en función del objeto que lo llame.
- Adquiere su máxima potencia cuando se utiliza junto con la **herencia** y las **clases abstractas**.
- Las reglas para utilizar el **polimorfismo** en Java son:

  → Crear una jerarquía de clases (**herencia**) definiendo las operaciones generales a través de **métodos abstractos** (o no) en la **clase base**.

  → Cada una de las clases derivadas debe **definir sus propias versiones** de los **métodos abstractos** y **sobrescribir** los métodos necesarios.

  → Los **objetos derivados** se manipularán a través de una **referencia** a la **clase base** para conseguir la **ligadura dinámica**.

Macarena Cuenca Carbajo

Métodos y clases no derivables

- Un **método no derivable** es aquél definido como **final**, lo que implica que el método no puede ser redefinido en **clases derivadas**.
- Ejemplo:

```
public class Ventana {
    final public int numPixeles() {...}
}

public class Ventana2 extends Ventana {
    public int numPixeles() {...}
}
```

- Cuando un método es definido como **no derivable** significa que no va a cambiar, con lo cual, el compilador puede colocar el *bytecode* del método en **tiempo de compilación**.
- Es decir, los métodos no derivables tienen **ligadura estática**.

Centro de Enseñanza Gregorio Fernández

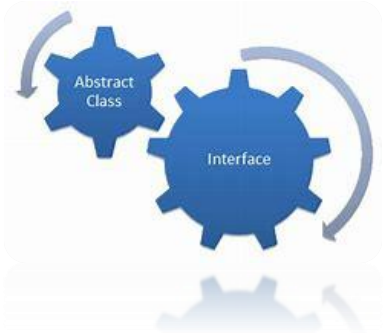
Métodos y clases no derivables

- Si queremos que una **clase** sea **no derivable** la declaramos **final**:

```
final public class Ventana {
...
}
```

- Al declarar la clase **Ventana final**, no puede tener descendencia.
- Las **clases no derivables** no van a formar parte de una jerarquía de clases, serán clases independientes.
- Ejemplos de clases no derivables en Java son los **Wrappers**: Integer, Double, Character,…
- Todos los métodos de una clase **final** son implícitamente **métodos no derivables**, aunque puede haber clases que tengan métodos **final** y otros que no lo sean.

Interfaces



- Una **interfaz,** al igual que una clase abstracta, sirve para definir el comportamiento mínimo de una seria de clases.
- La diferencia está, en que todos los métodos de una interfaz son **públicos** y **abstractos**.
- De hecho, no está permitido preceder a los métodos de modificadores de visibilidad diferentes de **public**.
- Una interfaz no puede tener constructor, con lo cual no se pueden crear objetos de una interfaz, aunque sí declarar referencias a la interfaz.
- Para definir una **interfaz** se hace de forma similar a la definición de una clase, pero sustituyendo la palabra **class** por **interface.**

Macarena Cuenca Carbajo

Interfaces

- Sintaxis:

```
public interface NombreInterfaz {
        constante1;
        constante2;
        ...
        constanteN;
        tipo metodo1 ([lista de argumentos]);
        tipo metodo2 ([lista de argumentos]);
        ...
        tipo metodoN ([lista de argumentos]);
}
```

Los atributos que contiene una interfaz, implícitamente son public, static y final.

Centro de Enseñanza
Gregorio Fernández

Interfaces
I - Implementación

- Las interfaces no se heredan, sino que se **implementan**.
- Para indicar que una clase implementa una **interfaz** se hace de forma similar a la herencia, pero sustituyendo la palabra **extends** por **implements**:

```
public class NombreClase implements Interfaz {
        //atributos
        //métodos de la clase
        //implementación de los métodos de la interfaz
}
```

- Las clases que implementen una **interfaz** están obligadas a **implementar todos sus métodos**, ya que sino la clase se convertiría en **abstracta**.
- Además, si la interfaz tiene constantes, será obligatorio darlas un valor.

Centro de Enseñanza
Gregorio Fernández

Interfaces
I - Implementación

- Ejemplo:

```
public interface Barco {
    void alarma();
    void msgSocorro(String msg);
}

public class BarcoPasaje implements Barco {

    private int numCamas;

    public BarcoPasaje(int n) {
       this.numCamas=n;
    }

    public void alarma() {
       System.out.println("Que no cunda el pánico");
    }

    public void msgSocorro(String msg) {
       this.alarma();
       System.out.println("¡SOS! " + msg);
    }
}
```

La implementación de los métodos de la interfaz se hace public, ya que java no permite reducir su visibilidad.

Centro de Enseñanza
Gregorio Fernández

Interfaces
I - Implementación

- Como sabemos Java no admite la **herencia múltiple**, pero sí que permite que una clase implemente varias interfaces:

```
public class NombreClase implements Interfaz1, Interfaz2,... {
        //atributos
        //métodos de la clase
        //implementación de los métodos de todas las interfaces
}
```

- Es una forma de simular la **herencia múltiple**.
- En este caso, la clase debe implementar todos los métodos de todas las interfaces.

Interfaces
II - Jerarquización

- Al igual que las clases, las **interfaces** se pueden organizar jerárquicamente, de forma que una **interfaz** pueda heredar los métodos de otra.
- Mientras que una clase sólo puede heredar de una **clase base** (herencia simple), una **interfaz** puede heredar de tantas interfaces como necesite.
- De la misma forma que en las clases, para indicar que una interfaz hereda de otra se utiliza la palabra reservada **extends**.
- Una clase puede heredar de otra clase e implementar una o varias interfaces. En primer lugar, se debe indicar la clase de la que hereda (**extends**) y posteriormente la/s interfaces que implementa (**implements**).

Interfaces
II - Jerarquización

- Ejemplo:


```
public interface Parque { … }
public class EspacioProtegido { … }
public class ZonaAves extends EspacioProtegido { … }
public class ParqueNatural extends EspacioProtegido
                           implements Parque { … }
```

Actividades

pdf Ejercicios UD.6