Módulo III

Herencia, Interfaces, Casting y Clases Anidadas

Contenido

- Herencia.
- Binding dinámico.
- Interfaces.
- Casting.
- Clases anidadas.

Herencia

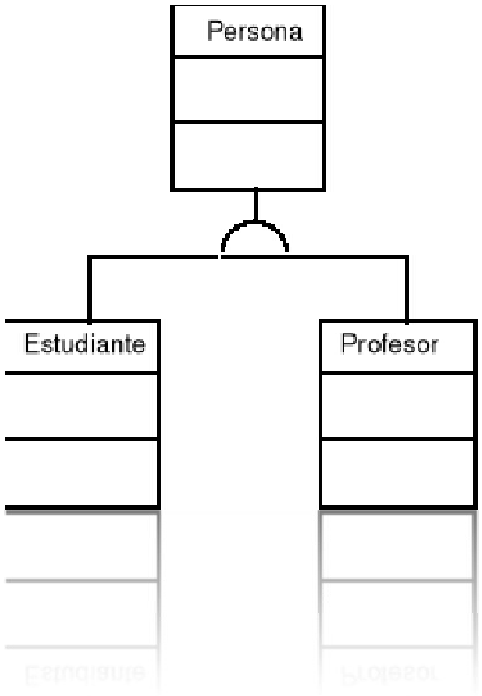
- Es un mecanismo de la programación orientada a objetos mediante el cual podemos reutilizar comportamiento de una clase “base” y extenderlo con comportamiento más específico.
- La subclase hereda de su super clase:
  - Estructura: variables de instancia
  - Comportamiento: métodos de la clase base.

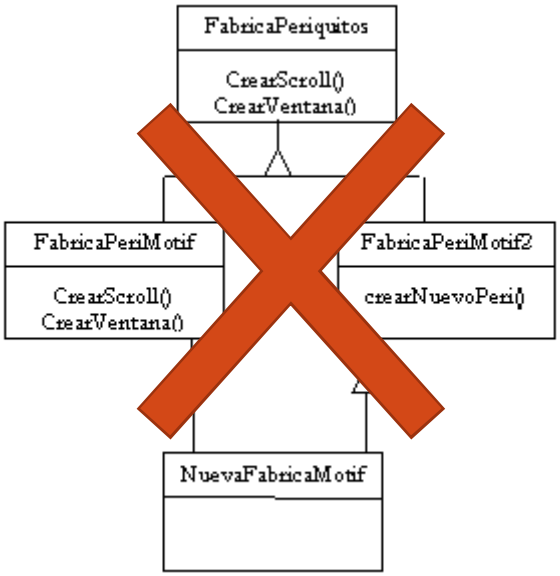
```
class <claseDerivada> extends <claseBase> {
...
}
```

```
class Circulo extends Figura {
...
}
```

Herencia

- Java implementa herencia simple





Herencia

- Comportamiento
- Variables

- Pero…
  - Existen algunas restricciones
    - Final
    - Private
    - Package

Se ve restringido el acceso

Herencia

- Toda clase es derivada de otra excepto Object que es la raíz.

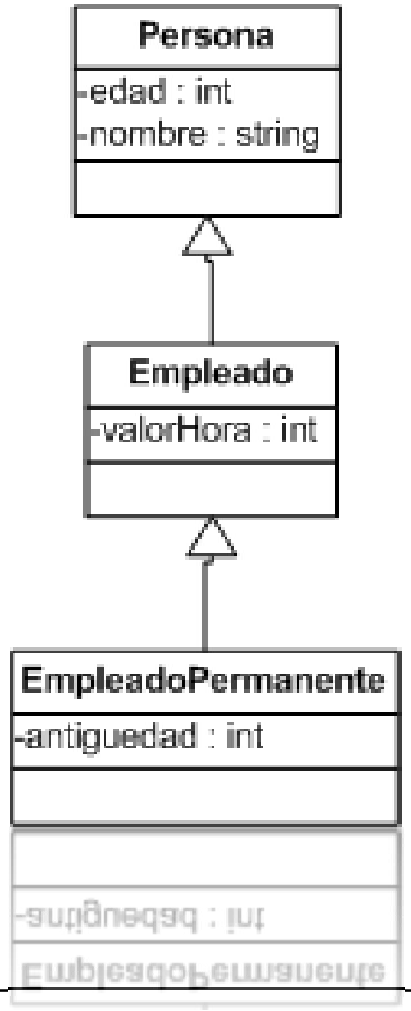
```
public class A{…}

public class B extends Object{…}

public class C extends Figura{…}
```

Constructores

- La forma que posee Java para crear instancias es invocando toda la cadena de constructores hasta llegar a Object.



•Al hacer new EmpleadoPermanente() Java realiza:

1. Implicitamente invoca al super constructor sin parámetros.
2. Al llegar a Object crea la parte básica del objeto y comienza a "bajar en las ejecuciones".
3. Persona realiza su parte de constructor y aporta la estructura.
4. Empleado su parte.
5. Empleado permanente finaliza.

Constructores

- En cada constructor por default se ejecuta como primera linea “super()”.
  - ¿Qúe sucede si la super clase no posee un constructor sin parametros?
  - ¿Qué sucede si la super clase posee un constructor sin parámetros pero es privado?
  - ¿Cómo puede hacer para invocar explícitamente a otro constructor?

Herencia – Modificadores

- En variables y métodos se utilizan : public, protected, private y packate (si no se define ninguno).

```
mod_visibibilidad <Tipo> nombre_variable;
...
mod_visibibilidad <Tipo_Retorno> nombre_metodo(params){...}
```

```
private String nombre;
Int catidad; //es protected
...
public String getNombre(){...} //es public
int calculoInterno(){...}
```

Modificador **Abstract**

- Un método abstracto es declarado pero no implementado:
  - No tienen cuerpo (implementaciones en las subclases).
- Una clase abstracta está compuesta de uno o más métodos abstractos.
  - Puede tener también métodos no abstractos.
  - Si una clase tiene un método abstracto, tiene que ser declarada también abstracta.
- Restricciones de una clase abstracta:
  - Se puede tener métodos constructores.
  - Los métodos abstractos no pueden especificarse estáticos.
  - Métodos privados no pueden especificarse como abstract.
  - Una subclase de una clase abstract debe implementar todos los métodos abstract heredados o en su defecto también debe ser declarada como abstract.

Métodos abstract

- Un método abstracto se declara pero no posee implementación
  - public abstract void imprimir(String texto);
- Si una clase posee un método abstracto, la clase debe ser declarada abstracta.
- Los métodos declarados abstractos no pueden ser declarados final.

Clases Abstractas - Herencia

- Consecuencias de sub clasificar a una clase abstracta
  - Si la súper clase posee métodos abstractos entonces:
    - Si la subclase NO es abstracta, entonces debe implementar todos los métodos abstractos que hereda.
    - Si decide implementar solamente algunos y otros no, entonces la subclase debe ser abstracta también. ¿ Por qué ?

Modificadores de accesibilidad - **Final**

- El modificador **final** especifica**:**
  - Para una clase, que esta no puede extenderse.
  - Para un método, que este no puede redefinirse.
  - Para una variable de instancia, que esta tiene un valor constante (su valor no puede ser modificado).
- La herencia queda “inhibida” por el uso de este modificador.
- Para cada clase, variable o método afectado por este modificador, se puede decir que es la "versión final".
- Si un método no es afectado por este modificador (que limita su posibilidad de sobrescritura), se dice que el método es “virtual” (por defecto).
- ¿Cuándo usar el modificador final?

Modificadores de accesibilidad - Conclusión

- Como conclusión:
  - Los métodos declarados con el modificador **abstract** en la superclase **deben** ser sobrescritos en las subclases.
    - Para las variables no tiene sentido.
  - Los métodos declarados con el modificador **final** en la superclase **no pueden** ser sobrescritos.
  - Las variables declaradas con el modificador **final** en cualquier clase, tienen un valor constante una vez asignadas.

Herencia - Constructores

- En Java, los constructores no se heredan.
- Una subclase solo puede invocar a los constructores de su super (usando super) clase y de ella misma (usando this).
- Si el constructor de la super clase requiere algún parámetro, este deberá ser explicitado:

```
class Empleado extends Persona{
public Empleado(String nombre, float basicoHora){
    super(nombre);//invocación del constructor de la clase base
    ...
    }
...
}
```

Herencia - Métodos

- Un método puede ser redefinido en una subclase por diversos motivos:
  - Performance.
  - Agregar alguna funcionalidad adicional (template method).
  - ...
- Un método de una clase base definido nuevamente en una clase derivada se considera *sobrescrito*, o *redefinido* u ***overriden*** (sólo en ausencia de *final* y *abstract*).

```
Public class Empleado extends Persona{
...
public String toString(){
return super.toString() + “precio de hora trabajada: ” +
                          this.getPrecioHora();
} //se invoca al método sobrescrito }
```

Métodos heredados de object

```
public boolean equals(Object anObject) {
     if (this == anObject) {
          return true;
     }
     if (anObject instanceof String) {
        //comparo variables internas
     return true;
          }
     }
return false;
  }
```

- `public String toString()`
  - Se utiliza para obtener una representación textual de los objetos.
  - Los wrappers de tipos primitivos (Integer, Character, Boolean) lo sobrescriben.
  - El compilador lo utiliza automaticamente por ejemplo en el el System.out.println() o en la conversión a String de un objeto.

```
String var = “el objeto es ” +
objetoX;
```

Herencia – Variables

- Las variables no pueden sobrescribirse. Si una subclase redefina una variable con el mismos nombre que la heredada conviven y puden ser accedidas.

```
public class ClaseBase{
    int x = 1;
}
public class ClaseDerivada extends ClaseBase{
    int x = 2;
public void metodo(){
    System.out.println(this.x); //2
        System.out.println(this.x); //2
        System.out.println(super.x); //1
    }
}
```

Polimorfismo y overloading

- La **sobrecarga** de métodos es un tipo de polimorfismo llamado ad-hoc.
- Se dice que un método está sobrecargado cuando existe otro con el mismo nombre, pero con diferentes implementaciones.
- Se caracteriza por determinarse en tiempo de compilación y por lo general se da por diferencia en los tipos de los parámetros.

```
public class Punto{
public Punto suma(int x, int y){...}
public Punto suma(Punto p){...}
}
```

Polimorfismo

- La sobre escritura es otra relación de polimorfismo.
- Ocurre sólo en un contexto de relación clase base – clase derivada.
- También existen dos definiciones del mismo método, sin embargo la signatura debe ser la misma o equivalente.

```
public class Persona{
…
        public String toString(){…}
}
public class Empleado{
        …
        public String toString(){…}
}
```

Interfaces

- Una interfaz consiste en una colección de signaturas de métodos y declaraciones de constantes agrupadas bajo un nombre.
- En una interfaz se indica qué se hace, pero no cómo se hace. Definen un protocolo de comportamiento que puede ser implementado por cualquier clase.
- Una clase puede extender a varias interfaces.

```
package nombrePaquete;

{imports}

[public] interface NombreInterfaz [extends ListaDeSuperInterfaces] {
  [Constantes]
  [Signaturas de métodos]
}
```

Interfaces

- Ejemplos:

```
public interface Observer {
    /**
     * This method is called whenever the observed object is changed. An
     * application calls an <tt>Observable</tt> object's
     * <code>notifyObservers</code> method to have all the object's
     * observers notified of the change.
     *
     * @param   o     the observable object.
     * @param   arg   an argument passed to the <code>notifyObservers</code>
     *                method.
     */
    void update(Observable o, Object arg);
}
```

Interfaces

```
interface InstrumentoMusical{
        void sonar();
    int LA = 440;
}



public class Cello implements InstrumentoMusical {

        @Override
        public void tocar() {
            System.out.println("Tocamos en " + LA);

}

}
```

Interfaces - Conclusiones

- Una interfaz es un protocolo de comportamiento. Puede ser implementada por cualquier clase, perteneciente a cualquier jerarquía.
- Una interfaz evita establecer relaciones forzadas entre clases para compartir una superclase abstracta.
- Las interfaces no deben crecer. Si se cambia el comportamiento de una interfaz, todas las clases que la implementan fallarán.
- Una clase puede implementar más de una interfaz.
- Java provee herencia múltiple de interfaces.
- El mayor beneficio obtenido es potenciar polimorfismo en un sistema tipado.

Interfaces

```
interface ColPrimarios{
    int ROJO=1, VERDE=2,
AZUL=4;
}
```

```
interface TodosLosColores
   extends ColImpresion,
ColArcoIris {
   int FUCSIA=17, BORDO=ROJO+90;
}
```

```
interface ColArcoIris extends
ColPrimarios {
    int AMARILLO=3,
NARANJA=5, INDIGO=6,
VIOLETA=7;
    public void unMetodo():
}
```

```
interface ColImpresion extends
ColPrimarios {
   int AMARILLO=8, CYAN=16,
MAGENTA=32;
   public void unMetodo();
}
```

```
public class MisColores
implements
ColImpresion,ColArcoIris {

  public MisColores() {
      int unColor = AMARILLO;
  }
  public void unMetodo(){
    //algo
  }
}
```

OK

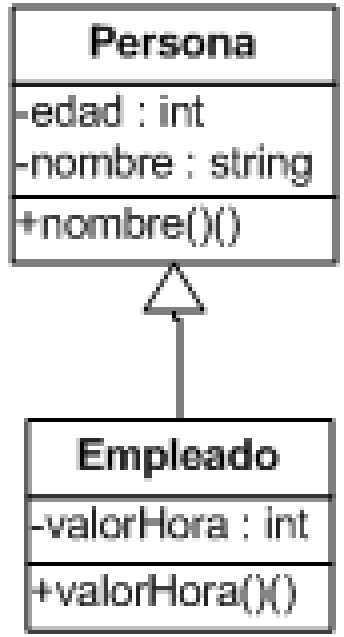
No se especifica cual constante se utiliza

Casting

- El casting consiste en covertir el tipo de una expresión en otro.
- Puede ser implícito
  - if (3>'a') {}
  - String hola = “hola”;
  - Object eraHola = hola;
- El casting puede hacerse en el mundo de las clases e interfaces.
- También entre tipo primitivos.
- NO puede hacerse entre tipos primitivos y clases e interfaces.
  - **char** a;
  - Object x = (Object) a;

Upcasting

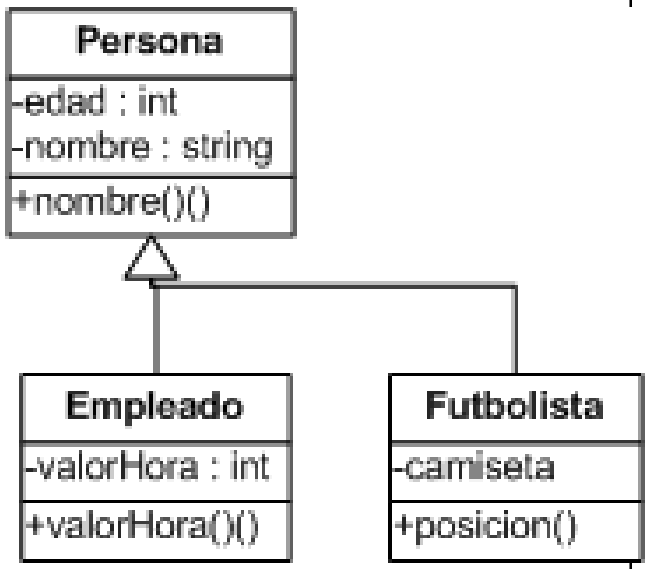
- Consiste en convertir la expresión en alguna de sus super tipos.
- Es seguro, pero se pierde la especialización (comportamiento).
- El upcasting es en general implícito.




```
Empleado diego = new Empleado("Diego", 5);
Persona p = (Persona) diego
p.nombre();
p.valorHora();
```

Downcasting

- Consiste en convertir la expresión en una de un tipo más específico.
- NO siempre es seguro.
- Pueden generar errores en ejecución:




```
ClassCastException
Object[] array = {new Empleado("Diego"),
                  new Futbolista("Julian")};
...
for (i = 0; i < array.length(); i++){
   String s = (Empleado)array[i];
}
```

Casting

- También se puede hacer casting de un tipo a otro, si el segundo es una interface implementada en el primero.

```
class MyClass implements MyInterface…
…
Object[] array…

MyInterface a = (MyInterface) arr[3];
```

Clases anidadas (Inner classes)

- Una clase anidada es una clase que está definida dentro de otra.

```
class UnaClase {
  . . .
  class UnaClaseAnidada {
    . . .
  }
}
```

Clases anidadas (Inner classes)

- Características:
  - Sólo pueden instanciarse dentro de su clase contenedora.
  - Se instancian de la misma forma que una clase común.
  - Puede haber múltiples niveles de clases anidadas.
  - Las clases anidadas pueden tener las mismas características que las clases comunes, salvo que no pueden declarar ni métodos ni variables **`static`**.