Sesión 20: Herencia 2/2

Programación 2

Ángel Herranz

2021-2022

Universidad Politécnica de Madrid

En capítulos anteriores

- 👍 Tema 1: Intro a POO
- 👍 Tema 2: Clases y Objetos y TADs y módulos
- 👍 Tema 3: Colecciones con *arrays*
- 👍 Tema 4: Cadenas simplemente enlazadas
- 🕒 Tema 5: TAD Listas
- 🕒 Tema 7: Herencia[1] y Polimorfismo
  - ***Ad hoc***: mencionaremos sobrecarga
  - **Genéricos**: polimorfismo paramétrico
  - **Subtipado**: herencia

[1] Sólo de interfaz.

En el capítulo de hoy

- Tema 7: Herencia y Polimorfismo
  - **Subtipado**: conversión, LSP[2]

[2] *Liskov Substitution Principle*

Herencia

- Las instancias de una subclase heredan todas las *propiedades* (atributos y métodos) de la superclase
- En Java el *enlazado* de los métodos se realiza en tiempo de ejecución[3]
- En las subclases se pueden sobreescribir las propiedades

"Un gran poder..."

[3] *Dynamic Dispatching* o *Dynamic binding* vs. *Static binding*

Object en Java

Todas las clases herenda de Object

- Si una clase no declara **extends**

```
public class Animal {...}
```

- por defecto hereda de `Object`
- Es como si se hiciera

```
public class Animal extends Object {...}
```

¿Y qué metodos se heredan de `Object`

Consultar la documentación de `Object`

¿Y qué metodos se heredan de `Object`

Consultar la documentación de `Object`

```
public boolean equals(Object obj)
public String toString()
```

¿Y qué metodos se heredan de `Object`

Consultar la documentación de `Object`

```
public boolean equals(Object obj)
public String toString()
protected Object clone()
public int hashCode()
```

¿Y qué metodos se heredan de `Object`

Consultar la documentación de `Object`

```
public boolean equals(Object obj)
public String toString()
protected Object clone()
public int hashCode()
protected void finalize()
public Class<?> getClass()
```

¿Y qué metodos se heredan de `Object`

Consultar la documentación de `Object`

```
public boolean equals(Object obj)
public String toString()
protected Object clone()
public int hashCode()
protected void finalize()
public Class<?> getClass()
public void wait()
public void notify()
```

Conversiones de tipo

Java nos ayuda con la herencia i

```
public class Mamifero extends Animal {...}
```

```
public static Mamifero f() {...}
```

```
Animal x = f();
```

Java nos ayuda con la herencia i

```
public class Mamifero extends Animal {...}
```

```
public static Mamifero f() {...}
```

```
Animal x = f();
```

Java compila 👍

Java nos ayuda con la herencia ii

```
public class Mamifero extends Animal {...}
```

```
public static Animal f() {
  return new Mamifero();
}
```

```
Mamifero x = f();
```

Java nos ayuda con la herencia ii

```
public class Mamifero extends Animal {...}
```

```
public static Animal f() {
  return new Mamifero();
}
```

```
Mamifero x = f();
```

Java no compila 👍

Conversión de tipos

- ¡El código estaba bien! ¿Por qué no compila?
- Podemos *fozar* al compilador a que lo entienda con una conversión de tipos[4]

```
public class Mamifero extends Animal {...}
```

```
public static Animal f() {
  return new Mamifero();
}
```

```
Mamifero x = (Mamifero) f();
```

[4] *Type casting*, *Type coercion*

Downcasting i

```
public class Mamifero extends Animal {...}
public class Pez extends Animal {...}
```

```
public static Animal f() { return new Pez();}
```

```
Mamifero x = (Mamifero) f();
```

Downcasting i

```
public class Mamifero extends Animal {...}
public class Pez extends Animal {...}
```

```
public static Animal f() { return new Pez();}
```

```
Mamifero x = (Mamifero) f();
```

Java compila y se rompe en ejecución

 ¡Probadlo!

Downcasting ii

```
public class Mamifero extends Animal {...}
```

```
public static Animal f() {
  return new Mamifero();
}
```

```
Mamifero x = (Mamifero) f();
```

Downcasting ii

```
public class Mamifero extends Animal {...}
```

```
public static Animal f() {
  return new Mamifero();
}
```

```
Mamifero x = (Mamifero) f();
```

- Pero como yo sé que no se rompe, se lo quiero decir al compilador mediante un *downcasting*
- !Esto **no** es una buena idea!

instanceof i

```
public class Mamifero extends Animal {...}
```

```
public static Animal f() {...}
```

```
Mamifero x = null;
if (f() instanceof Mamifero) {
  x = (Mamifero)f();
}
```

instanceof ii

- Y ya no se rompe aunque `f()` devuelva un pez

```
public class Mamifero extends Animal {...}
public class Pez extends Animal {...}
```

```
public static Animal f() { return new Pez();}
```

```
Mamifero x = null;
Animal a = f();
if (a instanceof Mamifero)
  x = (Mamifero) a;
assert x == null;
```

instanceof iii

```
/** Método que lee una figura */
public Figura leerFigura() {...}
```

```
/** Imprimir la longitud del lado */
Figura fig = leerFigura();
if (fig instanceof PoligonoRegular) {
  System.out.println(
    "Lado: "
    +                    fig .lado()
  );
}
```

instanceof iii

```
/** Método que lee una figura */
public Figura leerFigura() {...}
```

```
/** Imprimir la longitud del lado */
Figura fig = leerFigura();
if (fig instanceof PoligonoRegular) {
  System.out.println(
    "Lado: "
    + ((PoligonoRegular)fig).lado()
  );
}
```

Herencia de interfaz

Herencia de interfaz

- Muchos lenguajes permite describir interfaces[5]
- Un interfaz declara un conjunto de métodos (API) que luego las clases tendrán que implementar
- Nos va a recordar a las clases abstactas
- Pero se usan más y con más riqueza que las clases abstractas: herencia múltiple

[5] También llamados *traits* en algunos lenguajes

CRUD: un interfaz de persistencia

```
public interface CRUD {
  boolean create(Data d);
  Data    read(Id id);
  boolean update(Data d);
  boolean delete(Id id);
}
```

- Es *como* una clase abstracta donde todos los métodos son abstractos
- No hay atributos, ni constructores, sólo interfaz
- Todo es público

¿Para qué sirve? i

```
public class Controller {
  private CRUD storage;
  public Controller (CRUD storage) {
    this.storage = storage;
  }
  public void onSave(Data d) {
    if (storage.update(d))
      notifySaveDone();
    else
      notifySaveFailed();
  }
  ...
}
```

¿Para qué sirve? i

```
public class Controller {
  private CRUD storage;
  public Controller (CRUD storage) {
    this.storage = storage;
  }
  public void onSave(Data d) {
    if (storage.update(d))
      notifySaveDone();
    else
      notifySaveFailed();
  }
  ...
}
```

- Se pueden usar como si fueran un tipo (como las clases)
- ¿Podemos programar sin conocer la implementación?

¿Para qué sirve? ¡ Más ocultación

```
public class Controller {
  private CRUD storage;
  public Controller (CRUD storage) {
    this.storage = storage;
  }
  public void onSave(Data d) {
    if (storage.update(d))
      notifySaveDone();
    else
      notifySaveFailed();
  }
  ...
}
```

- Se pueden usar como si fueran un tipo (como las clases)
- ¿Podemos programar sin conocer la implementación?

Dos implementaciones

```
public class DB
  implements CRUD {
  public DB(String host) {
    ...
  }
  public boolean create(Data d) {
    ...
  }
  public Data read(Id id) {
    ...
  }
  boolean update(Data d) {
    ...
  }
  boolean delete(Id id) {
    ...
  }
}
```

```
public class RestClient
  implements CRUD {
  public RestClient(String endpoint) {
    ...
  }
  public boolean create(Data d) {
    ...
  }
  public Data read(Id id) {
    ...
  }
  boolean update(Data d) {
    ...
  }
  boolean delete(Id id) {
    ...
  }
}
```

¿Para qué sirve? ii

- Hoy

```
public class App {
  public static void main(String[] args) {
    CRUD storage = new DB("postgresql://localhost:5432");
    Controller saveController = new Controller(storage);
  }
}
```

¿Para qué sirve? ii

- Hoy

```
public class App {
  public static void main(String[] args) {
    CRUD storage = new DB("postgresql://localhost:5432");
    Controller saveController = new Controller(storage);
  }
}
```

- Mañana

```
public class App {
  public static void main(String[] args) {
    CRUD storage = new RestClient("http://localhost:8080");
    Controller saveController = new Controller(storage);
  }
}
```

¿Para qué sirve? ii Más reusabilidad

- Hoy

```
public class App {
  public static void main(String[] args) {
    CRUD storage = new DB("postgresql://localhost:5432");
    Controller saveController = new Controller(storage);
  }
}
```

- Mañana

```
public class App {
  public static void main(String[] args) {
    CRUD storage = new RestClient("http://localhost:8080");
    Controller saveController = new Controller(storage);
  }
}
```

Herencia múltiple

```
public class Controller
  implements InputViewListener
             SaveViewListener {
  private Data data;
  public void onChanged(Data d) {
    data = d;
  }
  public void onSave() {
    storage.update(data);
  }
}
```

- Java no admite herencia múltiple entre clases (**extends**)
- Pero admite herencia múltiple de interfaces (**implements**)
- Masivamente usado para arquitecturar aplicaciones

LSP: “Un gran poder. . .”

LSP: *Liskov substitution principle*

Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T

Barbara Liskov (1987) and Jeannette Wing, 1994

LSP: Principio de Substitución de Liskov

Sea $\phi(x)$ una propiedad demostrable de los objetos x de tipo T. Entonces $\phi(y)$ debe cumplirse para los objetos y de tipo S si S sea subtipo de T

Barbara Liskov (1987) and Jeannette Wing, 1994

LSP: una buena práctica

Sea $\phi(x)$ una propiedad demostrable de los objetos x de tipo T. Entonces $\phi(y)$ debe cumplirse para los objetos y de tipo S si S sea subtipo de T

- Los compiladores nos protegen un poco
- Pero nos permiten reemplazar los métodos
- Así podríamos destruir las propiedades conceptuales de las superclases

👎 Ej. en `Cuadrado` podríamos cambiar `perimetro()` para que calcule el área (iríamos en contra del principio)

Últimas palabras sobre la herencia

- Herencia múltiple: difícil
- *Dynamic vs Static dispatching (binding)*
- *Scandinavian vs. American schools* (LSP o no LSP, flexibilidad o no flexibilidad)
- Principios SOLID:
  - **S**RP: Single-responsibility
  - **O**CP: Open-closed (extension-modification)
  - **L**SP: Liskov substitution[8]
  - **I**SP: Interface segregation (many)
  - **D**IP: Dependency inversion