Nociones necesarias de POO

Codificación en un Lenguaje

Introducción a la POO

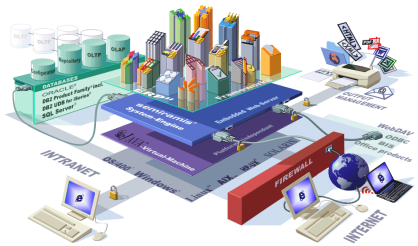
Clases, Objetos, Métodos y Atributos

Codificación en el Lenguaje

La complejidad del Software

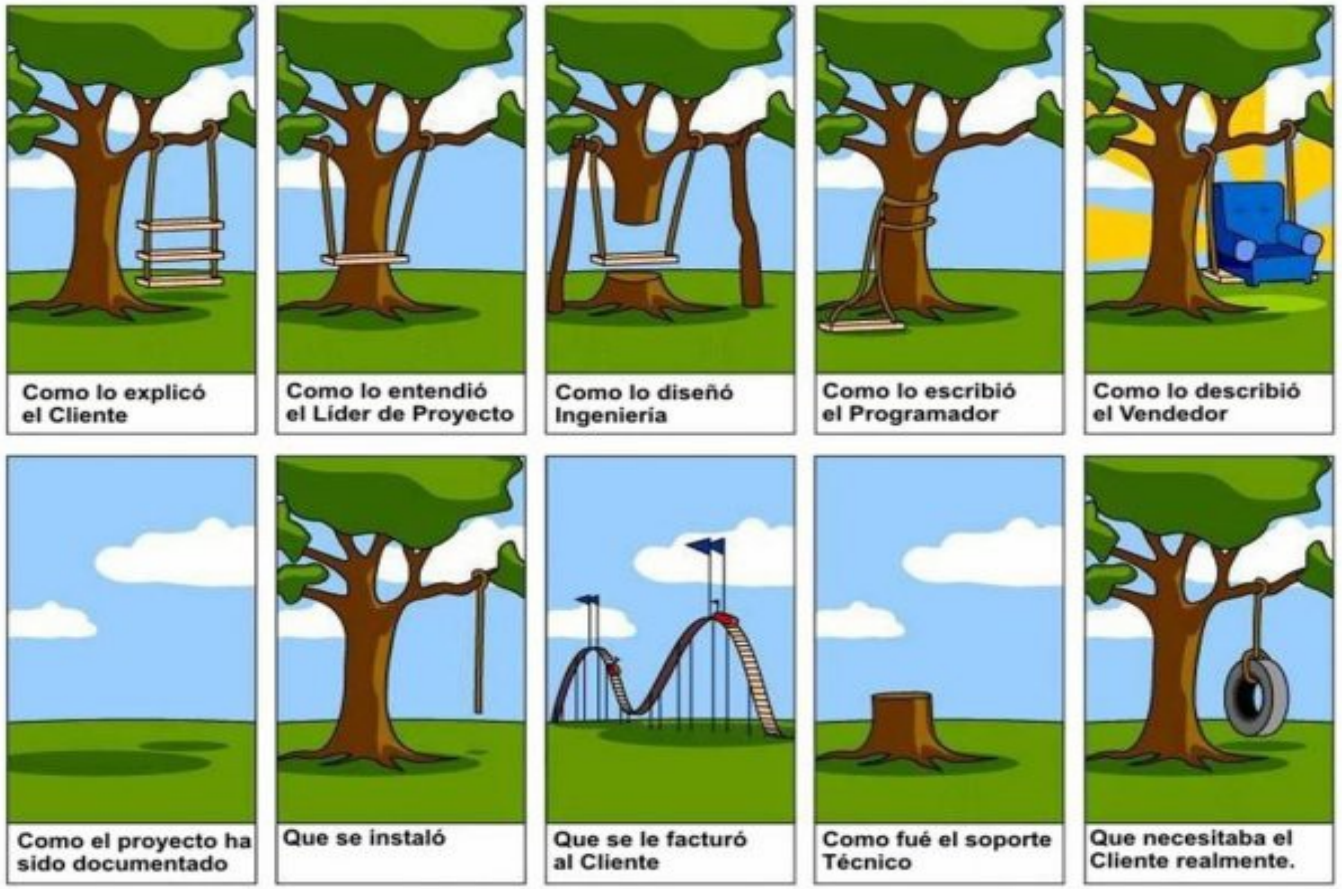
Es una propiedad esencial del software, depende 4 partes:

- El *dominio* del problema → Requisitos muy distintos, cambiantes, contradictorios.
- El *proceso* de desarrollo → Intervienen muchos desarrolladores, se necesita coordinar y utilizar una tecnología en común.
- La *flexibilidad* en el software → Sistema apto para cambiar con los requerimientos del cliente
- La *reutilización* del código → No reinventar la rueda.



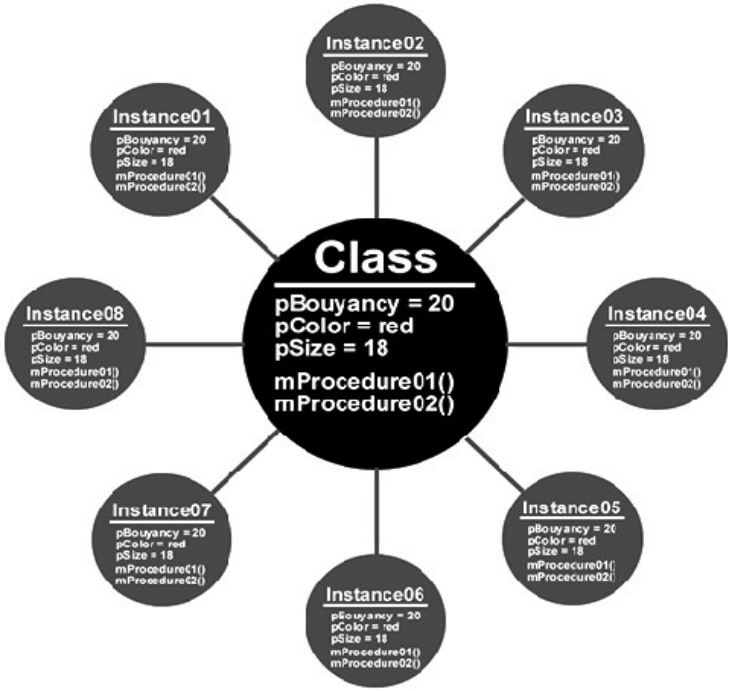
La complejidad del Software

Una triste realidad de los proyectos de Software...



Programación Orientada a Objetos

Es un paradigma de programación, es decir una forma de *analizar, diseñar* y *realizar soluciones.*



Objetivos:

- Escribir software fácilmente modificable y escalable.
- Escribir software reusable.
- Disponer de un modelo natural para representar un dominio.

Ventajas:

- Fomenta la reutilización del software.
- El software desarrollado es más flexible al cambio.
- Es más cercano a pensar a la forma de las personas.

Objetos y Clases

Clases

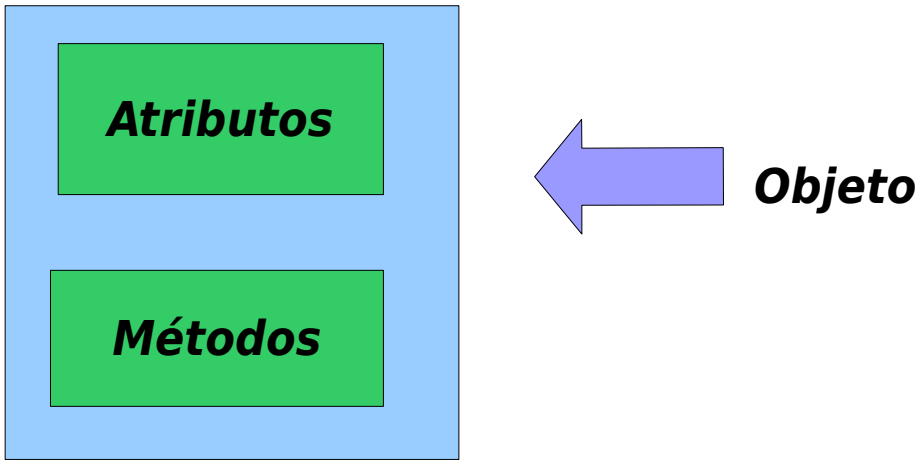
- Modela una abstracción de los objetos.
- Define atributos y comportamientos.
- Es el plano o molde que define un objeto.

Objetos

- Está modelado *en función* de una clase.
- Es una *instancia* única de una clase.
- Retiene la estructura y el comportamiento de una clase.

¿Qué es un Objeto?

- Es una entidad que combina procedimientos e información.
- Ejecuta operaciones (***comportamiento***) y almacena información (***estructura***).
- Los procedimientos o acciones que puede realizar un objeto se denominan ***métodos***.
- La información que almacena un objeto sobre su estado se denomina ***atributos***.



Los datos describen el *estado del objeto*.

Los procedimientos actúan sobre los datos del objeto, *modificándolos* o *brindando* información

¿Qué es un Objeto?

Un objeto posee...

Estado → Esta dado por el conjunto de propiedades que posee y su valor en un momento dado.

Comportamiento → Está dado por un conjunto de acciones que el mismo puede realizar.

Identidad → En todo momento es diferenciable del resto y es constante.

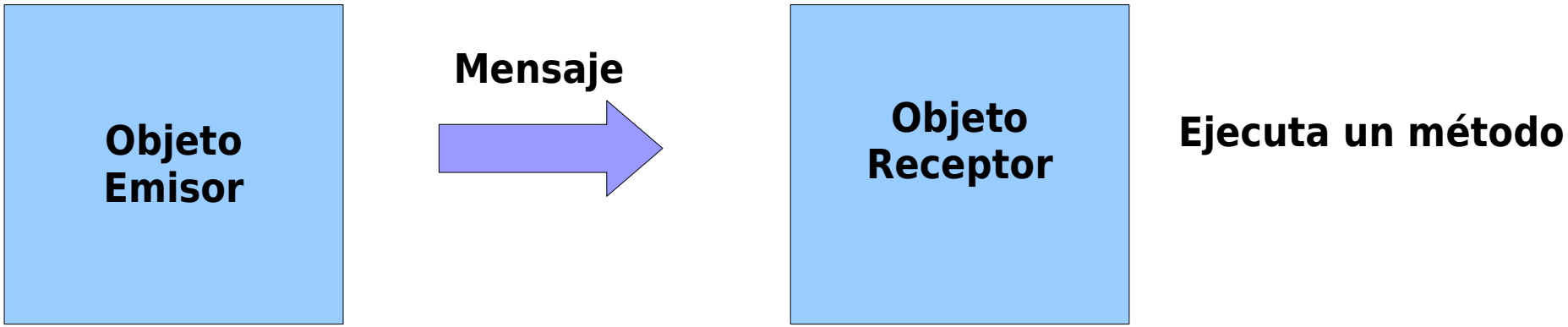
Comportamiento de un Objeto

- Determina como éste actúa y reacciona en términos de cambios de estado y solicitudes.
- Representan la interfaz por la cual se puede interactuar con el objeto.
- Los métodos asociados a un objeto comprenden el protocolo del mismo.

- Atributos:
  - color
  - velocidad
  - ruedas
  - motor
- Métodos:
  - arranca()
  - frena()
  - dobla()

Los objetos interactúan a través de mensajes

- Los objetos se comunican mediante el envío de mensajes.
- El objeto emisor se debe ***enlazar*** o ***asociar*** al objeto receptor.
- El emisor del mensaje solicita al receptor que realice una operación.
- La llamada al método siempre se produce en el contexto de un objeto concreto.



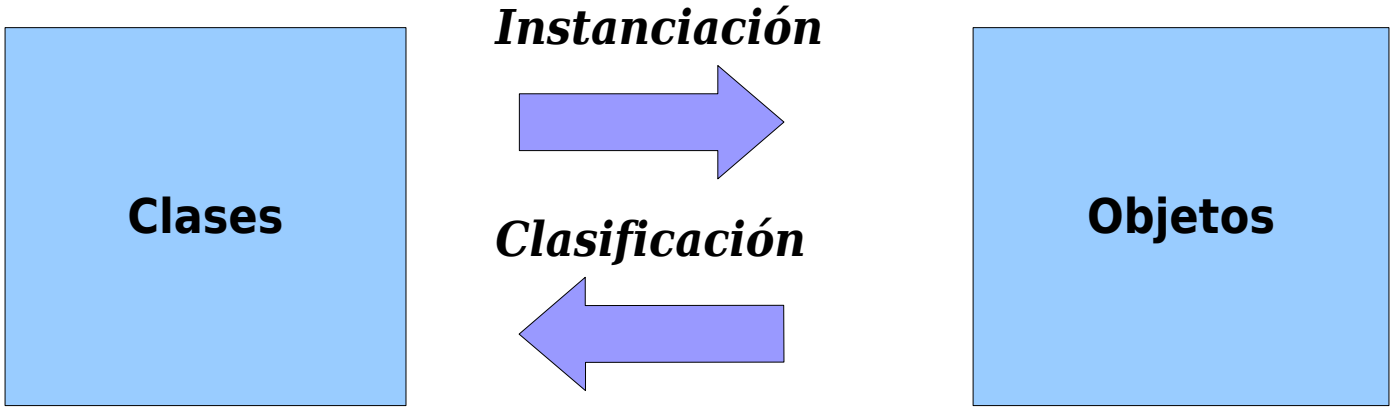
¿Qué es una Clase?

- Mecanismo utilizado en la POO para abstraer conceptos (clasificación).
- Describe las características comunes a todos los objetos que pertenecen a ella.
- Especifica:
  - El ***comportamiento*** de los objetos.
  - La ***estructura interna*** de los objetos.
  - La definición e implementación de todas sus ***acciones***.

Diferencias entre Clases y Objetos

- Las clases son ***definiciones estáticas*** que se pueden utilizar para entender a todos los objetos de una clase

- Los objetos son ***entidades dinámicas*** que existen en el mundo real.



- Definen atributos.
- Definen métodos.
- Generan instancias

- Poseen valores.
- Ejecutan métodos.

Definición de Clases

- Utilizaremos la palabra reservada ***class*** para definir una clase.
- Generalmente la definición de una clase está formada por:

Modificador de Acceso	Especifica la disponibilidad de esa clase.
Campos de instancia	Definen los atributos de la clase.
Constructores	Métodos con el mismo nombre de la clase, controlan su estado inicial.
Métodos de Instancia	Definen las acciones que las instancias de la clase pueden realizar.
Campos de clase	Atributos compartidos por todas las instancias de la clase.
Métodos de clase	Métodos utilizados para controlar los atributos de clase.

Definición de Clases

Un ejemplo...

```
public class Usuario {

    private String nombre;
    private String password;
    public static int cantidadLogins;

    public Usuario() {

    }


    public String getNombre() {
        return nombre;
    }


    public void setNombre(String nombre) {
        this.nombre = nombre;
    }


    public String getPassword() {
        return password;
    }
```

Definición de la clase

Variables de instancia y de clase

Constructor de la clase

Métodos de Instancia

Definición de Métodos

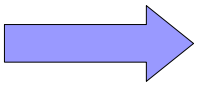
- Siempre se especifican dentro de una clase.
- Generalmente la definición de un método está formada por:

Modificador de Acceso	Especifica desde donde puede invocarse el método
Palabra clave static	Especifica que el método es estático.
Tipo de Retorno	Especifica el tipo de valor que el mismo debe devolver.
Argumentos	Parámetros necesarios para que el método realice su función
Nombre	Nombre del método

Definición de Métodos

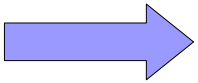
Algunos ejemplos...

```
public void setApellido(String apellido) {
    this.apellido = apellido;
}
```



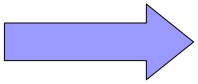
Método para establecer una propiedad

```
public void mostrarNombre() {
    String nombreCompleto = nombre + " , " +  apellido;
    System.out.println(nombreCompleto);
}
```



Método sin retorno (void)

```
public static int getCantidadLogins() {
    return cantidadLogins;
}
```



Método estático

Definición de Atributos

- Siempre se especifican dentro de una clase.
- Generalmente están definidos por:

Modificador de Acceso	Especifica desde donde puede verse el atributo.
Palabra clave static	Especifica que el atributo es estático.
Tipo de dato	Especifica el tipo de dato del atributo.
Nombre	Nombre con el cual lo identificaremos.

Definición de Atributos

Algunos ejemplos...

```
private long dni;
```

```
private String nombre;
```

```
private Date fechaNacimiento;
```

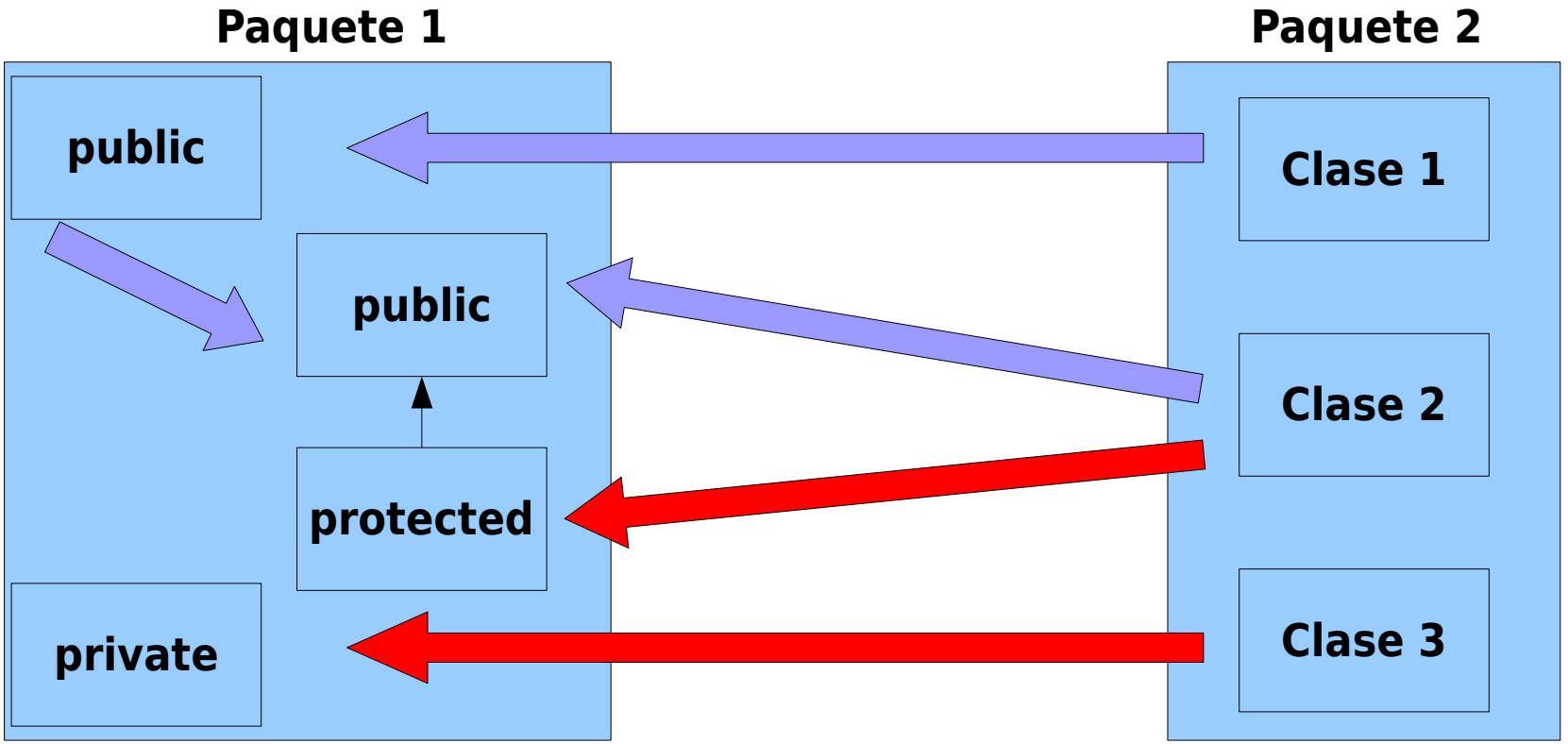
Atributos privados

```
public static boolean cargaHabilitada = true;
```

Atributo público

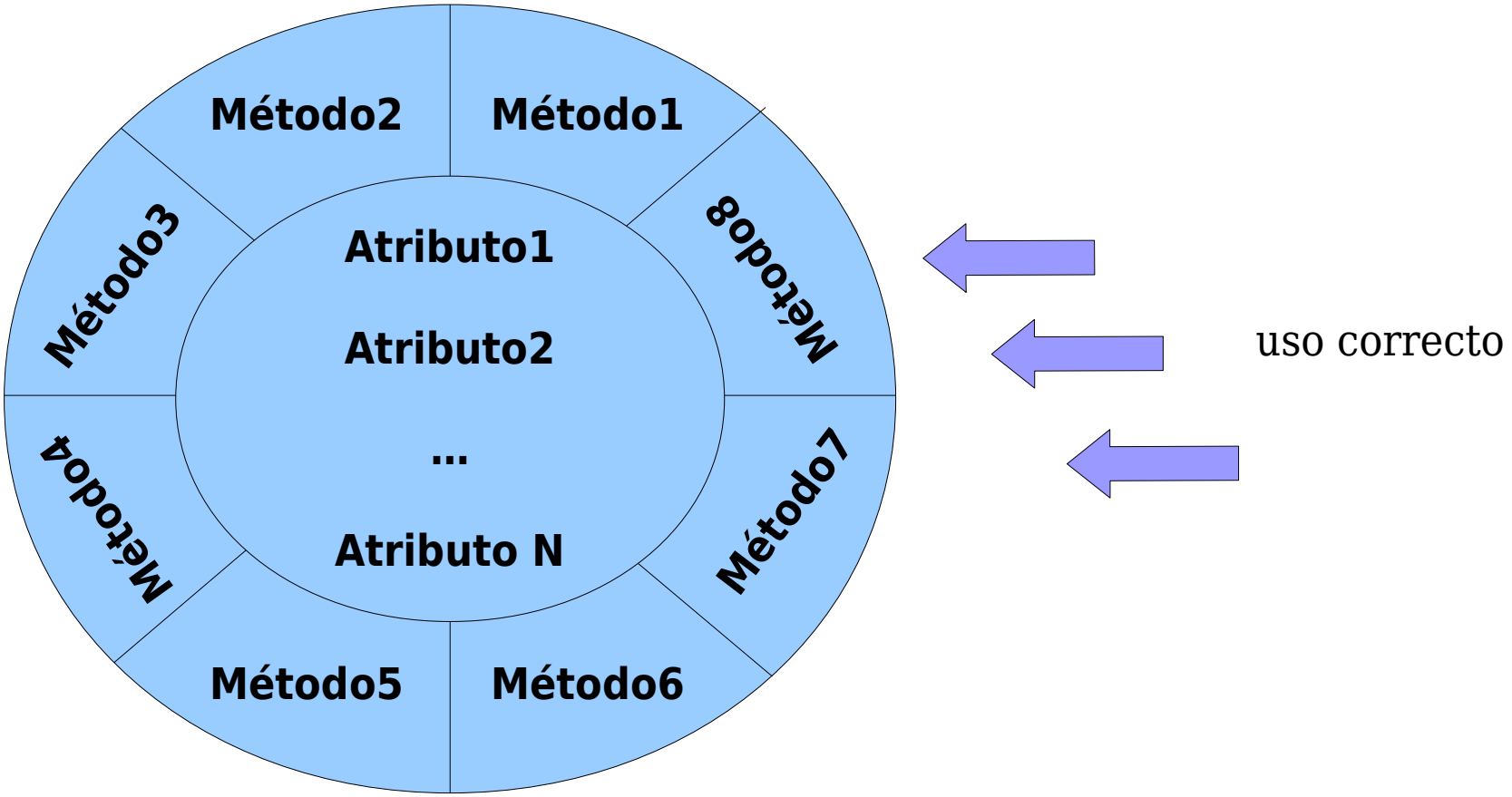
Modificadores de Acceso

- Controlan el acceso a los atributos y métodos.
- Para utilizar la encapsulación se recomienda utilizar variables privadas y métodos públicos.



Encapsulamiento

- Es una propiedad que asegura que la información de un módulo este oculta al exterior.
- Ocultando el estado de un objeto, solamente podremos modificar sus atributos mediante sus métodos.



Modificadores de Acceso

- Existen 3 tipos a definir, ***public*** – ***protected*** – ***private***.
- Si no especificamos ninguno se utiliza default.

Accesible a	***Public***	***Protected***	***Default***	***Private***
Misma clase	Sí	Sí	Sí	Sí
Cualquier clase del mismo paquete	Sí	Sí	Sí	No
Clase hija de un paquete diferente	Sí	Sí	No	No
Cualquier otra	Sí	No	No	No

Métodos get (obtener) y set (establecer)

Setters

- Cambian el valor de un atributo.
- Comienzan con *set* seguido del nombre del atributo.
- Ejemplos:

```
void setApellido(String apellido) { ... };
void setEdad(Date edad) { ... };
void setNombre(String nombre) { ... };
```

Getters

- Obtienen el valor de un atributo.
- Comienzan con g*et* seguido del nombre del atributo.
- Ejemplos:

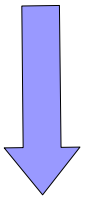
```
String getApellido() { ... };
Date getEdad() { ... };
String getNombre() { ... };
```

Uso de la referencia “this”

- Puede utilizarse dentro del cuerpo de un método de un objeto.
- Lo utilizamos para evitar la ambigüedad entre nombres de atributos o métodos.

```
public void setNombre(String nombre) {
    this.nombre = nombre;
}
```

Variable *nombre* utilizada como parámetro.



this.nombre especifica la propiedad nombre de la clase actual.

- También podemos utilizarlo para invocar a métodos o constructores.

```
String nom = this.getNombre();
```

Constructores

- Métodos con el mismo nombre de la clase.
- Se utilizan para controlar el estado inicial en el que se crea un objeto.
- No poseen valor de retorno (ni siquiera void).
- No se pueden invocar directamente como otro método.

```
public Usuario() {

}
```

Constructor vacío

```
public Usuario(long dni, String nombre, Date fechaNacimiento) {
    super();
    this.dni = dni;
    this.nombre = nombre;
    this.fechaNacimiento = fechaNacimiento;
}
```

Constructor con Parámetros

Creando de Objetos

Primeramente debemos crear una variable de referencia a ese objeto.

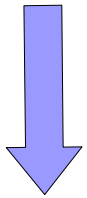
```
Usuario u;
```

Luego debemos inicializar esa referencia utilizando el operador *new* y llamando al constructor correspondiente:

```
u = new Usuario();
```

Constructor vacío

```
Usuario u = new Usuario(30164258, "nflores", new Date("25/05/1983"));
```



Constructor con Parámetros

Utilizando de Objetos

Existen dos formas de usar un objeto:

- Manipulando sus atributos

```
u.dni = 34629273;
```

No es recomendable, rompe la encapsulación

- Invocando sus métodos

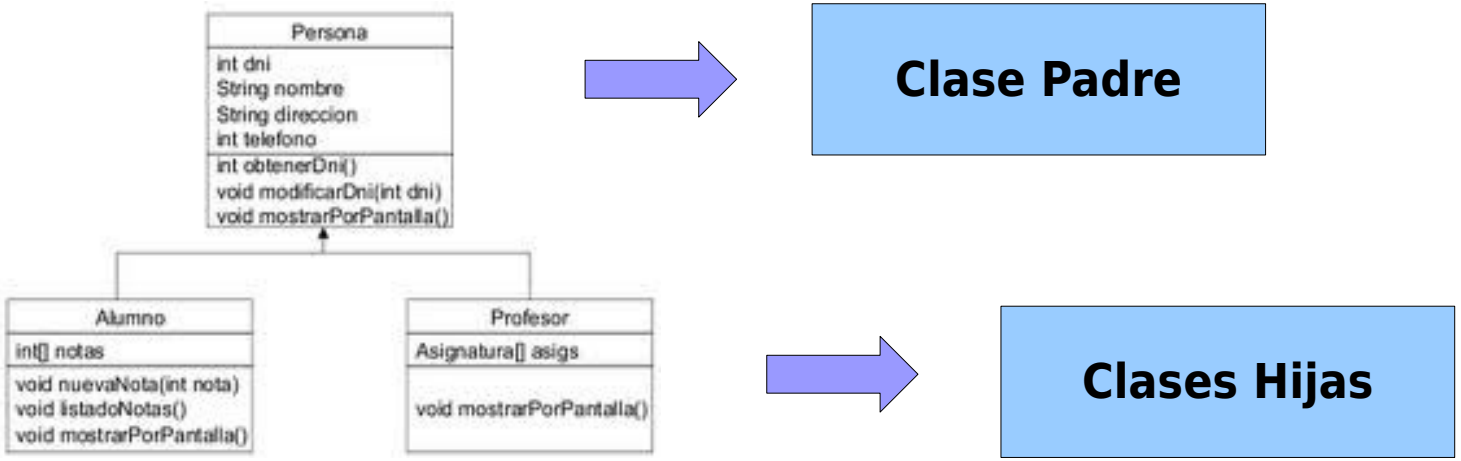
```
long l = u.getDni();
```

Utilizamos el operador “.”

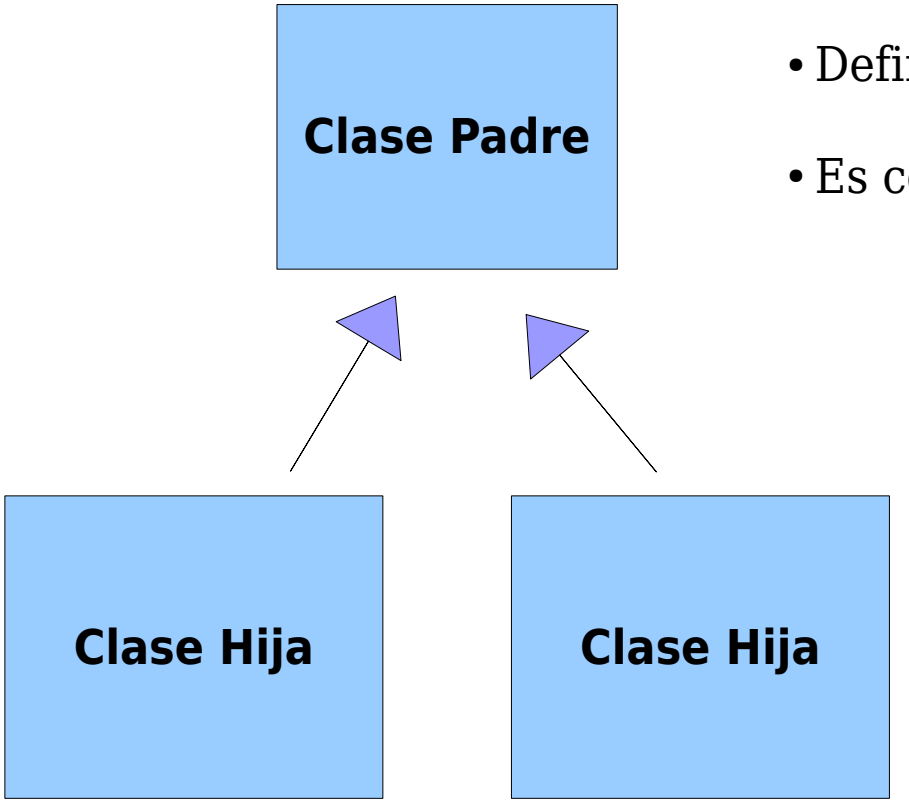
```
u.setNombre("invitado");
```

Herencia

- La reusabilidad puede lograrse mediante ***herencia***.
- Un comportamiento definido en una *superclase* es heredado por sus *subclases*.
- Las subclases extienden la funcionalidad heredada
- Esto nos permite definir la mayor cantidad de funcionalidades y atributos y luego reutilizarlas.



Herencia

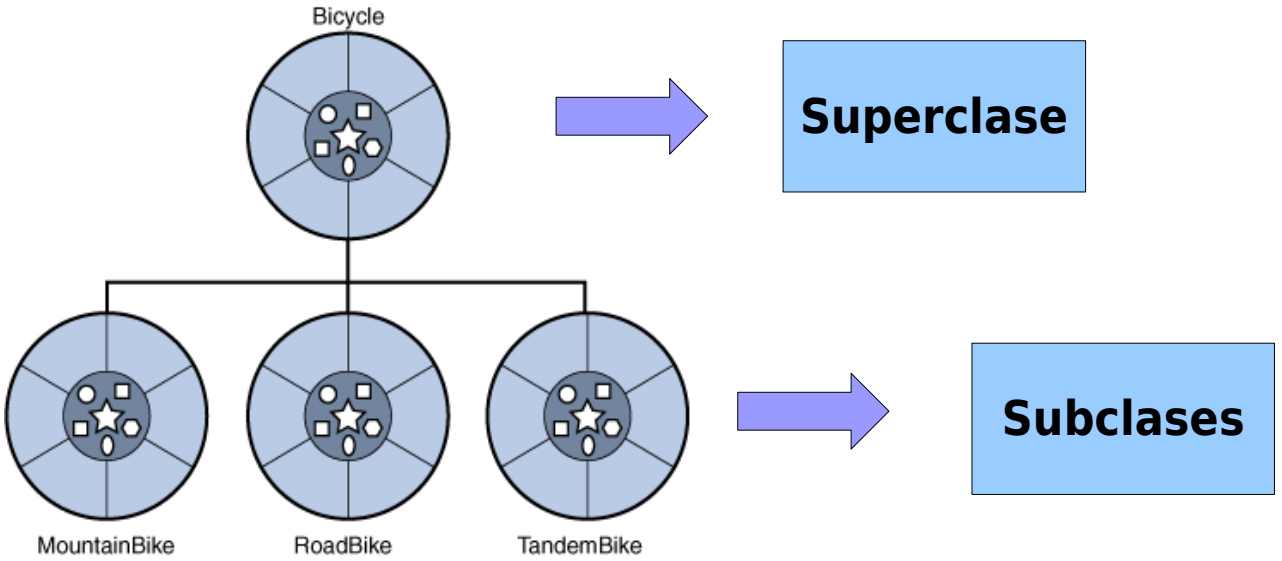


- Definimos atributos y comportamientos comunes.
- Es considerada como un tipo de generalización.

- Reutilizamos todo lo definido en la clase padre.
- Podemos ampliar atributos y comportamientos o *redifinirlos*.

Herencia

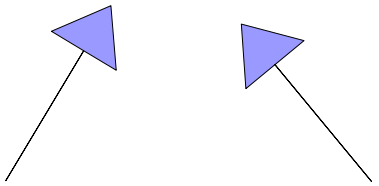
- Una clase hereda de su padre todos los atributos y métodos públicos y protegidos.
- Los constructores no son heredados pero pueden invocarse.
- Los métodos y atributos privados no se heredan.
- Si la subclase esta en el mismo paquete que la clase padre, también se heredan los miembros default.



Herencia en Java

- Para representar la herencia utilizaremos la palabra clave *extends*.
- Java no soporta la herencia múltiple.
- Si no especificamos herencia, todo extiende de la clase *Object*.

```
public class Persona {
}
```

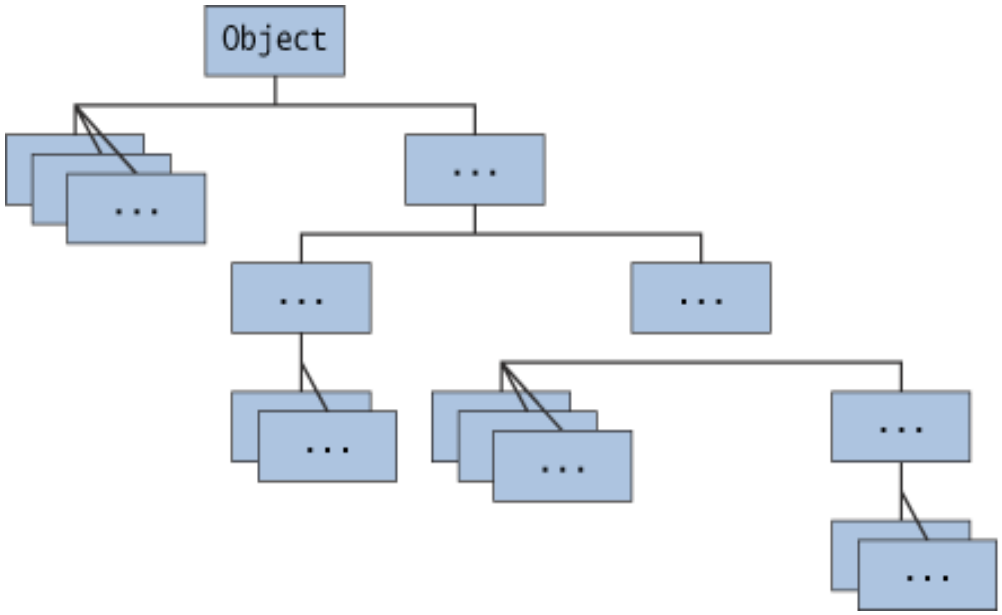


```
public class Profesor extends Persona {
}
```

```
public class Alumno extends Persona {

}
```

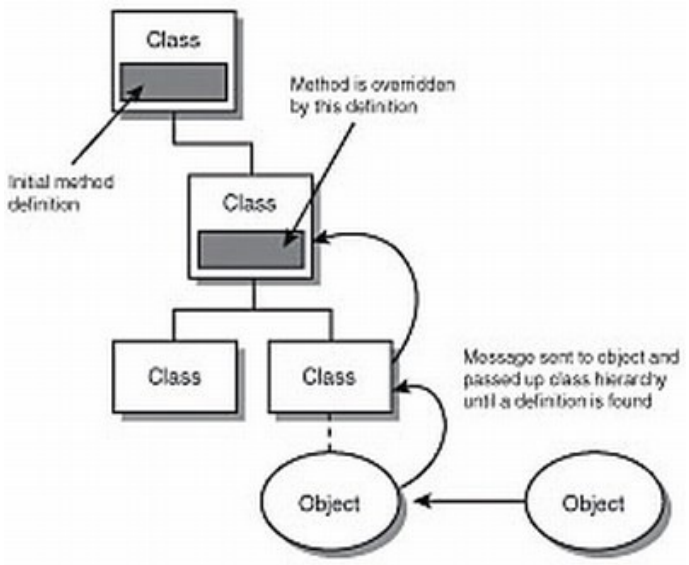
La clase Object

- Es la clase padre de toda clase en Java.
- Toda clase hereda implícitamente de ella.
- Provee métodos muy útiles para sobreescribir.
- Si una clase extiende directamente de ella no es necesario especificarlo



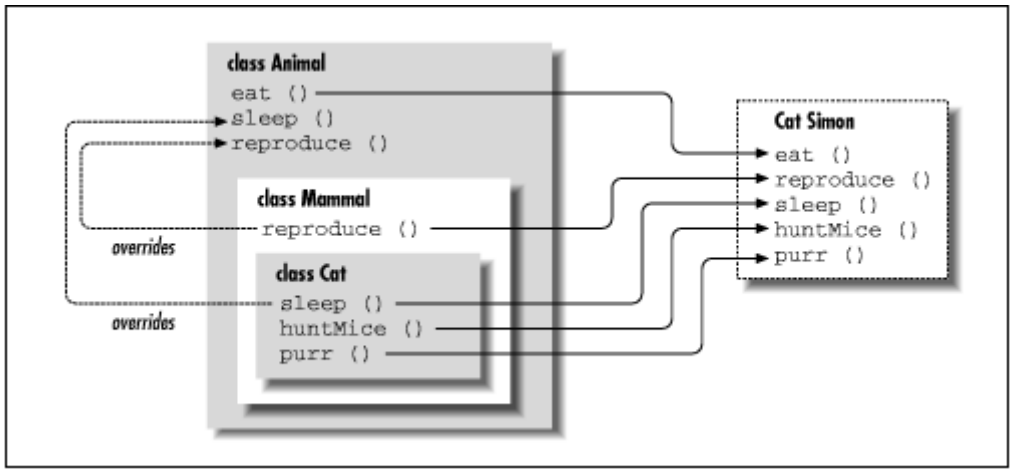
Sobreescritura de métodos

- Se da en el contexto de relaciones de herencia.
- Consiste en reescribir la implementación de un método de *instancia*, con su mismo nombre y argumentos (*firma del método*).
- Sólo pueden sobreescribirse los métodos no declarados como *final*
- Si una clase hija sobreescribe un método de su clase padre, entonces ocultará al mismo.



Sobreescritura de métodos

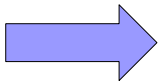
- La lista de argumentos debe ser idéntica en tipos y orden en ambos métodos.
- Los modificadores de acceso no pueden ser más restrictivos en la clase padre que en la subclase.
- Obtendremos un error de compilación si intentamos cambiar un método de instancia en una superclase a método de instancia en la subclase y viceversa.



Sobrecarga de métodos

- Dos o más métodos definidos en la *misma clase* y con igual nombre pueden sobrecargarse.
- Para ello es necesario que tengan el mismo nombre y tipo de retorno, pero *diferentes listas de argumentos (en número o tipos*)
- La JVM en tiempo real determina los parámetros con los cuales estamos invocando a un método y selecciona el correspondiente.

```
public class Artista {
    public void dibujar(int s) {
    }

    public void dibujar(String s) {
    }

    public void dibujar(int s, long y) {

    }
}
```



Métodos con el mismo nombre, pero con distinta firma.

Variable *super*

- Utilizada durante la sobreescritura, para acceder a métodos o atributos de instancia de la superclase ocultos para la subclase.
- Referencia a miembros de la superclase de la clase dada.

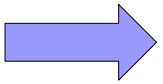
Referencia *this* → Apunta a atributos y métodos de la *clase actual*

Referencia *super* → Apunta a atributos y métodos de la *clase padre*

Variable *super*

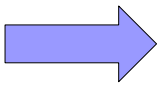
Un ejemplo

```java
public class Persona  {

    private String nombre;

    private String apellido;

    private Long edad;

    public void mostrarInfo() {
        System.out.println("Nombre: " + nombre);
        System.out.println("Apellido: " + apellido);
        System.out.println("Edad: " + edad);
    }
```



Clase padre que define el método *mostrarInfo()*

```java
public class Artista extends Persona {

    public void mostrarInfo() {

        super.mostrarInfo();

        System.out.println("Profesion: Artista");
    }
```

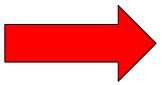


Clase que sobreescribe el método e invoca a su definición en la clase padre

El operador *instanceof*

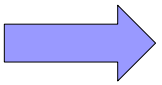
- Verifica que una referencia a un objeto sea de un tipo o de un subtipo dado.
- Podemos utilizarlo antes de convertir un objeto para evitar errores

```
Alumno alum = new Alumno("Esteban","Paz", 35L);

Profesor prof = (Profesor)alum;
```



Error de compilación

```
if (alum instanceof Persona) {
        Persona per = (Persona) alum;

}
```



Uso correcto

Alumno no es subclase de Profesor

Métodos y clases *final*

El calificador *final* condiciona el diseño de una jerarquía de herencia.

Clases Final:

- No pueden ser extendidas.

```
public final class Persona
```

Métodos Final:

- No pueden ser sobreescritos.
- Los métodos estáticos son automáticamente *final.*

```
public static final void getInfo() {

}
```

Se suele usar para evitar modificar una implementación

Métodos y clases *abstract*

El calificador *abstract* condiciona el diseño de una jerarquía de herencia.

Clases Abstract:

- No pueden ser instanciadas.

```
public abstract class Persona
```

Métodos Abstract:

- No pueden tener implementación.
- Deben ser implementados para las clases no abstractas que extiendan de su clase.

```
public abstract void trabajar();
```

Métodos y clases *abstract*

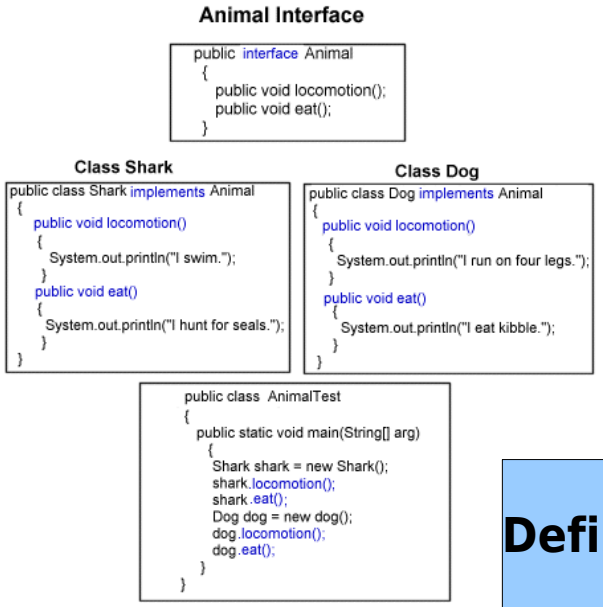
- Cuando un método está presente en todas las clases de una jerarquía pero se implementa en forma diferente conviene definirlo como abstracto.
- Una clase con al menos un método abstracto debe declararse como abstracta.
- Una clase que extiende de una clase abstracta debe:
  - Implementar todos sus métodos abstractos o …
  - Declararse como abstracta.

```java
public class Alumno extends Persona {

    public void trabajar() {
        System.out.println("Ir a la escuela");

    }

}
```

```java
public class Profesor extends Persona {

    @Override
    protected void trabajar() {
        System.out.println("Dar clases");

    }

}
```

Interfaces

- Bloques de códigos que poseen declaraciones de métodos y opcionalmente constantes.
- Todos sus métodos son abstractos, por ende no se definen.
- Una interface puede ser implementada por N clases.
- Una una clase implementa una interfaz se compromete a cumplir con su implementación, es decir se firma un contrato entre la clase y la interfaz.



Definimos interfaces utilizando la palabra clave *interface*

Interfaces

- Brindan la declaración de un comportamiento sin brindar la implementación.

```
public interface Imprimible {

    public void imprimir();

}
```

- Cada clase que se compromete a realizar una interfaz debe implementar todos sus métodos

```
public class DocumentoTexto implements Imprimible {

    public void imprimir() {
        System.out.println("imprimiendo el documento");
    }

}
```

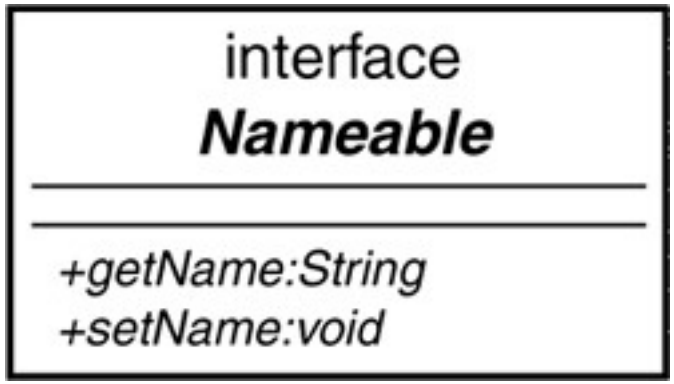
Interfaces

- Una interfaz puede contener:
  - Declaraciones de métodos públicos.
  - Constantes: atributos públicos, estáticos y finales.

- Las interfaces no pueden ser instanciadas.

- Si es pública debe coincidir con el nombre del archivo fuente (.java).

- Una referencia a una interfaz puede ser asignada a un objeto de cualquier clase que implemente esa interfaz.



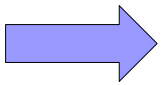
Interfaces

Para tener en cuenta:

La clase ***B*** extiende de la clase ***A*** e implementa la interface ***I***

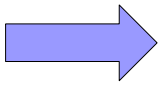
```
public class B extends A implements I {

}
```

```
B b = new B();
```



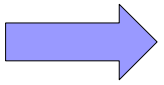
Se crea un objeto ***B*** referenciado por variable ***b*** de tipo de referencia ***B***

```
A a = new B();
```



Se crea un objeto ***B*** referenciado por variable ***a*** de tipo de referencia ***A***

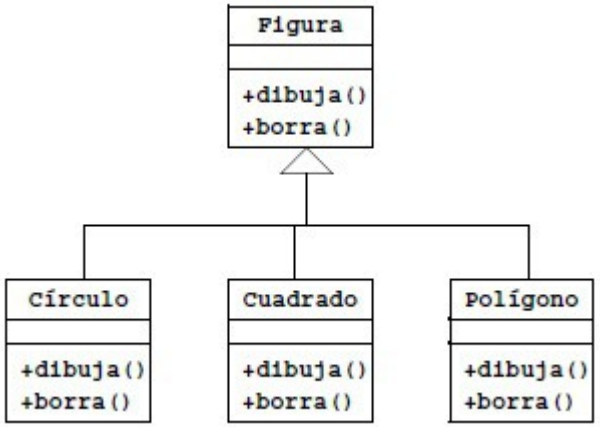
```
I i = new B();
```

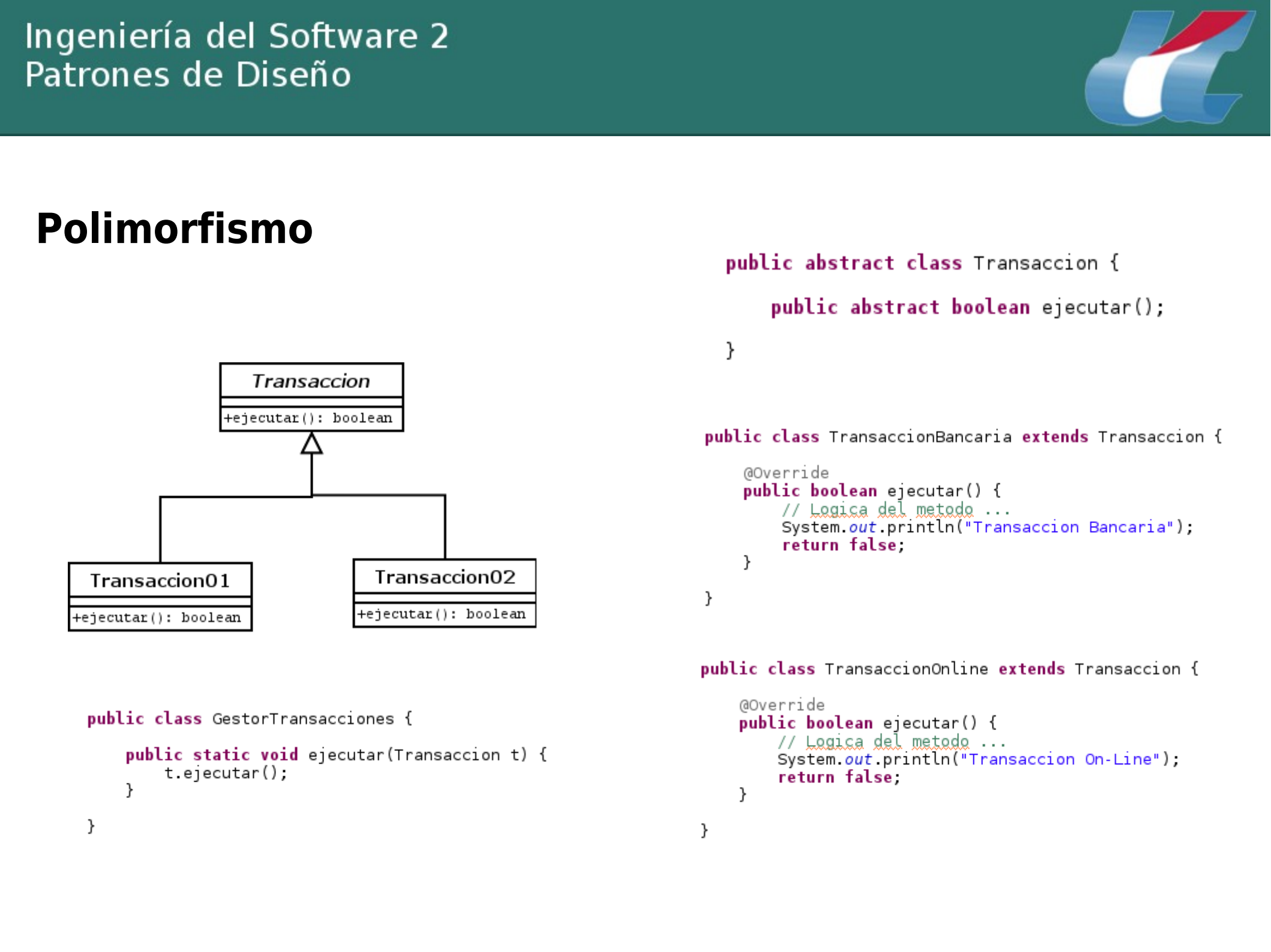


Se crea un objeto ***B*** referenciado por variable ***i*** de tipo de referencia ***I***

Polimorfismo

- Una variable de referencia cambia el comportamiento según el tipo de objeto al que apunta.
- Una variable traba a objetos de una clase como objetos de una superclase y se invoca dinámicamente el método correspondiente (*binding dinámico*)
- Si tenemos un método que espera como parámetro una variable de clase X, podemos invocarlo usando subclases pasando como parámetros referencias a objetos instancia de subclases de X.





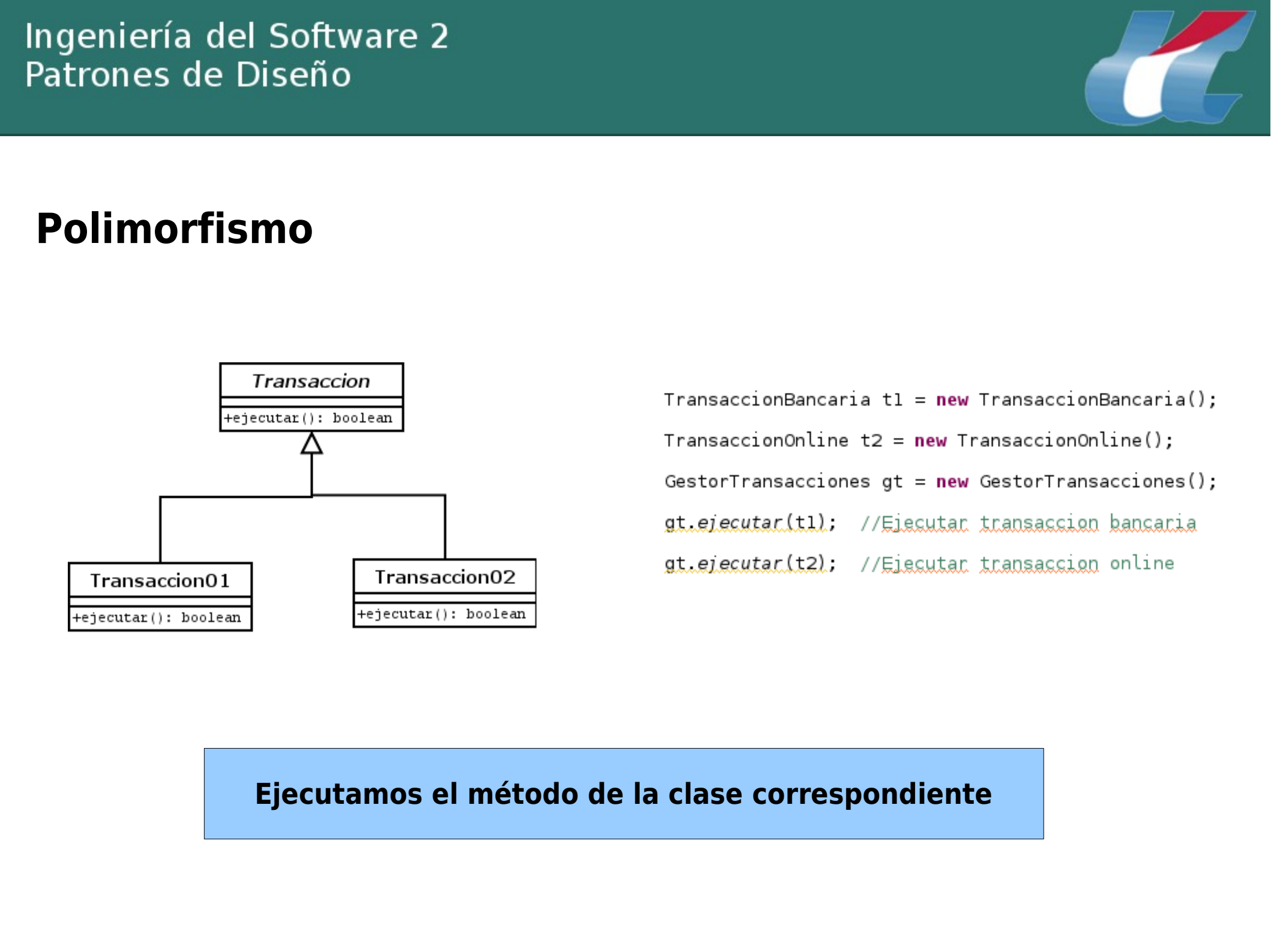
Polimorfismo


```java
public abstract class Transaccion {

    public abstract boolean ejecutar();

}
```

```java
public class TransaccionBancaria extends Transaccion {

    @Override
    public boolean ejecutar() {
        // Logica del metodo ...
        System.out.println("Transaccion Bancaria");
        return false;
    }

}
```

```java
public class TransaccionOnline extends Transaccion {

    @Override
    public boolean ejecutar() {
        // Logica del metodo ...
        System.out.println("Transaccion On-Line");
        return false;
    }

}
```

```java
public class GestorTransacciones {

    public static void ejecutar(Transaccion t) {
        t.ejecutar();
    }

}
```



Polimorfismo


```
TransaccionBancaria t1 = new TransaccionBancaria();
TransaccionOnline t2 = new TransaccionOnline();
GestorTransacciones gt = new GestorTransacciones();
gt.ejecutar(t1);   //Ejecutar transaccion bancaria
gt.ejecutar(t2);   //Ejecutar transaccion online
```

Ejecutamos el método de la clase correspondiente