



Ciencias de la
Computación

FACULTAD DE CIENCIAS
FÍSICAS Y MATEMÁTICAS
UNIVERSIDAD DE CHILE

CC3002 – Metodologías de
Diseño y Programación

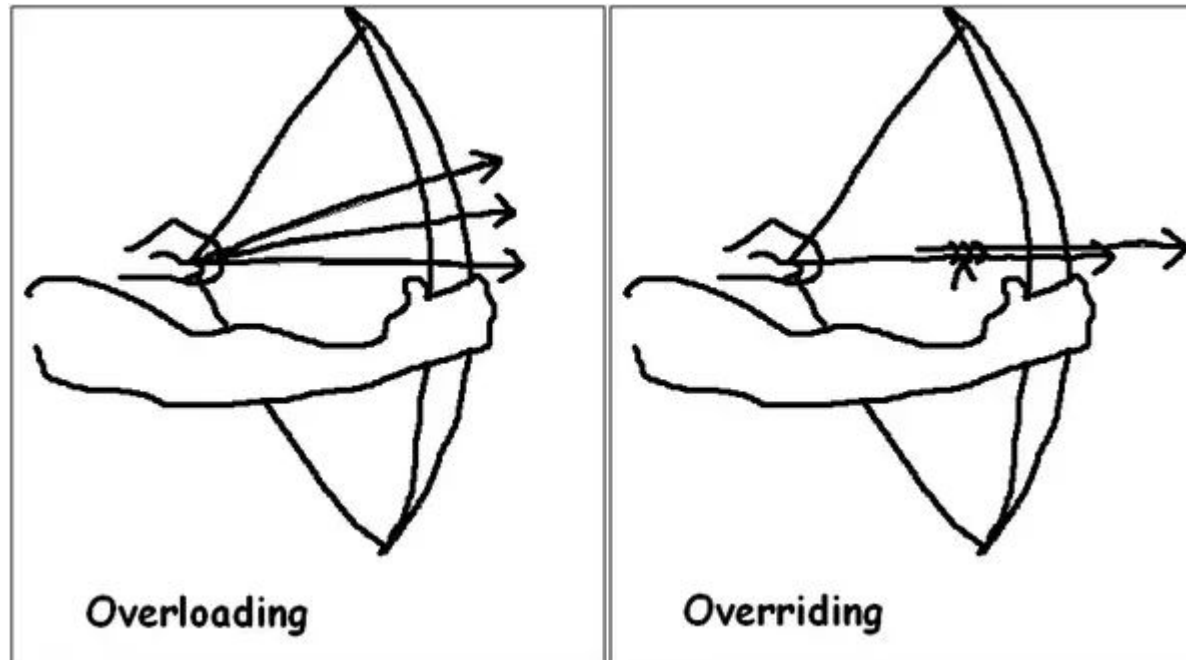
Auxiliar 2:

Constructores y herencia

Juan-Pablo Silva

jpsilva@dcc.uchile.cl

Overloading y Overriding



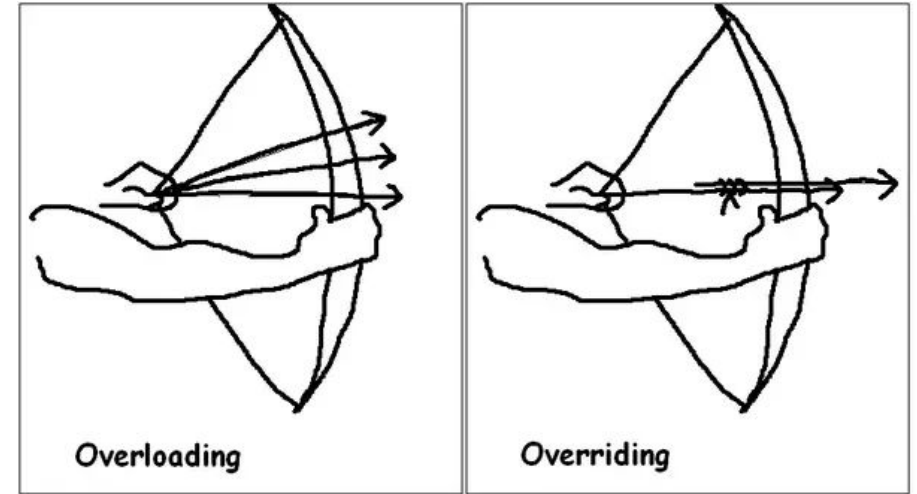
Overloading y Overriding

- Overriding:

- Métodos tienen la misma **firma**
- Define un comportamiento específico para la subclase
- Sobrescribe la funcionalidad

- Overloading:

- Permite tener más de un método con el mismo **nombre**
- Común: overloading de constructores
- Cuidado con los bugs



this y super (**IMPORTANTE**)

- this y super referencian al **mismo** objeto
 - ¿Cuál objeto?

this y super (**IMPORTANTE**)

- this y super referencian al **mismo** objeto
 - ¿Cuál objeto? El que recibe el mensaje (llamada)

this y super (**IMPORTANTE**)

- this y super referencian al **mismo** objeto
 - ¿Cuál objeto? El que recibe el mensaje (llamada)

¿En qué se diferencian entonces?

this y super (**IMPORTANTE**)

- this y super referencian al **mismo** objeto
 - ¿Cuál objeto? El que recibe el mensaje (llamada)
- Se diferencian en la forma en que comienza el *Method Lookup*

this y super (**IMPORTANTE**)

- this y super referencian al **mismo** objeto
 - ¿Cuál objeto? El que recibe el mensaje (llamada)
- Se diferencian en la forma en que comienza el *Method Lookup*
- Todo método que no tenga antepuesto *super*, tiene un *this* implícito
- La búsqueda en ambos casos sube por la jerarquía de clases en caso de no encontrar el método deseado, hasta llegar a Object

this y super (**IMPORTANTE**)

- this y super referencian al **mismo** objeto
 - ¿Cuál objeto? El que recibe el mensaje (llamada)
- Se diferencian en la forma en que comienza el *Method Lookup*
- Todo método que no tenga antepuesto *super*, tiene un *this* implícito
- La búsqueda en ambos casos sube por la jerarquía de clases en caso de no encontrar el método deseado, hasta llegar a Object
- *super* comienza el Method Lookup en... ¿Dónde?

this y super (**IMPORTANTE**)

- this y super referencian al **mismo** objeto
 - ¿Cuál objeto? El que recibe el mensaje (llamada)
- Se diferencian en la forma en que comienza el *Method Lookup*
- Todo método que no tenga antepuesto *super*, tiene un *this* implícito
- La búsqueda en ambos casos sube por la jerarquía de clases en caso de no encontrar el método deseado, hasta llegar a Object
- *super* comienza el Method Lookup en la superclase ¿De quién?

this y super (**IMPORTANTE**)

- this y super referencian al **mismo** objeto
 - ¿Cuál objeto? El que recibe el mensaje (llamada)
- Se diferencian en la forma en que comienza el *Method Lookup*
- Todo método que no tenga antepuesto *super*, tiene un *this* implícito
- La búsqueda en ambos casos sube por la jerarquía de clases en caso de no encontrar el método deseado, hasta llegar a Object
- *super* comienza el Method Lookup en la superclase de la **clase que contiene la llamada a *super***

this y super (**IMPORTANTE**) (pseudovariables)

- this y super referencian al **mismo** objeto
 - ¿Cuál objeto? El que recibe el mensaje (llamada)
- Se diferencian en la forma en que comienza el *Method Lookup*
- Todo método que no tenga antepuesto *super*, tiene un *this* implícito
- La búsqueda en ambos casos sube por la jerarquía de clases en caso de no encontrar el método deseado, hasta llegar a Object
- *super* comienza el Method Lookup en la superclase de la clase que contiene la llamada a *super*

*no olvidar

this y super (como constructores)

- Reutilizar constructores

```
public class A {  
    private int a;  
    private int b;  
  
    public A(int a, int b) {  
        this.a = a;  
        this.b = b;  
    }  
  
    public A() {  
        this(0, 0);  
    }  
}
```

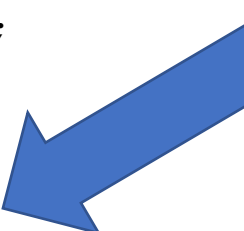
```
public class B extends A {  
    public B(int a, int b) {  
        super(a, b);  
    }  
  
    public B() {  
        // super()  
    }  
}
```

- Llamadas implícitas a super() (sin argumentos y en la primera línea!)

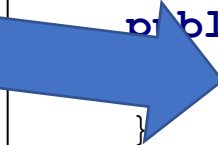
this y super (como constructores)

- Reutilizar constructores

```
public class A {  
    private int a;  
    private int b;  
  
    public A(int a, int b) {  
        this.a = a;  
        this.b = b;  
    }  
  
    public A() {  
        this(0, 0);  
    }  
}
```



```
public class B extends A {  
    public B(int a, int b) {  
        super(a, b);  
    }  
  
    public B() {  
        // super()  
    }  
}
```

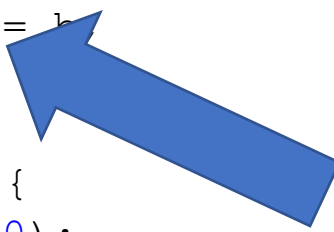


- Llamadas implícitas a super() (sin argumentos y en la primera línea!)

this (otros usos)


- Para desambiguar variables

```
public class A {  
    private int a;  
    private int b;  
  
    public A(int a, int b) {  
        this.a = a;  
        this.b = b;  
    }  
  
    public A() {  
        this(0, 0);  
    }  
}
```



- Como expresión para referenciarse a sí mismo

```
public B returnThis() {  
    return this;  
}
```



Polimorfismo (**IMPORTANTE**)

**La capacidad/habilidad de
un tipo A, de verse y poder
usarse como otro tipo B**

*No olvidar

Herencia e interfaces

- En java solo se puede extender una clase (directamente), pero se pueden implementar múltiples interfaces

Herencia e interfaces

- En java solo se puede extender una clase (directamente), pero se pueden implementar múltiples interfaces
- Una interfaz puede extender múltiples interfaces

Herencia e interfaces

- En java solo se puede extender una clase (directamente), pero se pueden implementar múltiples interfaces
- Una interfaz puede extender múltiples interfaces
- La reutilización de código es una consecuencia de la herencia, **nunca** el fin de esta. **NUNCA** usar herencia *solo* para reutilizar código

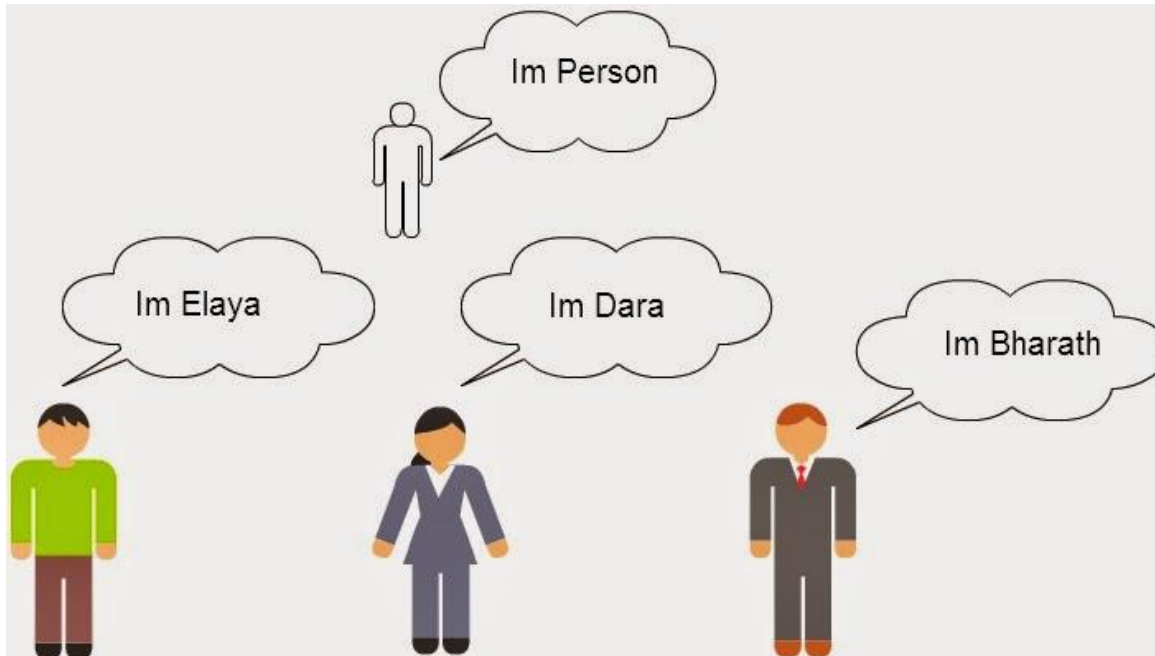
Herencia e interfaces

- En java solo se puede extender una clase (directamente), pero se pueden implementar múltiples interfaces
- Una interfaz puede extender múltiples interfaces
- La reutilización de código es una consecuencia de la herencia, **nunca** el fin de esta. NUNCA usar herencia **solo** para reutilizar código
- Los constructores **NO** se heredan

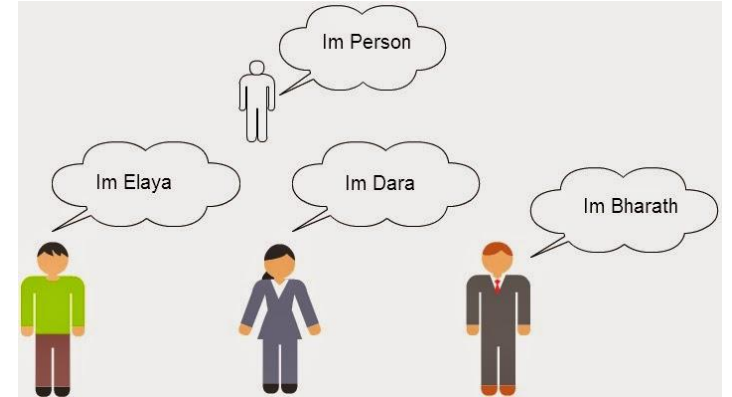
Clases abstractas vs Interfaces

- ¿Qué son?:

- CA) es una clase incompleta (no se puede/debe instanciar)
- I) es un contrato o garantía que doy a quien llame



Clases abstractas vs Interfaces



- ¿Qué son?:
 - CA) es una clase incompleta (no se puede/debe instanciar)
 - I) es un contrato o garantía que doy a quien llame
- ¿Cuándo se usan?:
 - CA) abstraer el funcionamiento genérico de una o más clases
 - I) nuevos tipos (polimorfismo)
- ¿Debe implementar todos los métodos? (en caso de ser una clase)
 - CA) sí, si no debe ser declarada clase abstracta
 - I) sí, si no debe ser declarada clase abstracta

Clases abstractas

NO LAS USEN COMO TIPOS

Clases abstractas

- Implementan un funcionamiento común entre todas las subclases
- Se pueden usar para definir comportamiento genérico/default y reemplazarlo en las clases concretas

Principio de Liskov



Principio de Liskov

*Los subtipos deben poder ser
sustituídos por sus clases padre*

Problemas



Estructuras de datos

- Mi abuela aún usa una agenda para guardar sus contactos.
- Anota el nombre y número de teléfono de la persona para no olvidarlo.
- Como buen nieto/a, quiero hacer un pequeño programa que cumpla la misma función: ¿Cómo lo haría?

Estructuras de datos

- El dueño del local de la esquina es conocido de mi papá y le dijo que estudiaba computación. Dice que tiene un gran problema para mantener el inventario, está haciendo todo en papel.
- ¿Qué solución propone para ayudarlo?

Estructuras de datos

- El mismo señor del local, ahora contento con su sistema de inventario, me comenta que aún escribe las boletas con las compras a mano.
- Lo ideal sería que de alguna forma se guardara lo que compró el cliente, para después poder revisarlo e imprimirlo automáticamente. ¿Cómo podríamos hacerlo?
- Si el cliente se arrepiente de algún producto, actualmente hay que romper la boleta en proceso y hacer otra.

Estructuras de datos

- Otra vez el señor del local...
- Escuchó el programa que le hice a mi abuela y ahora quiere una página web, donde con el número de la boleta, los clientes puedan ver lo que compraron. Esto ya lo se hacer, ¿o no?

Method Lookup

```
public class A {
    public String method1() {
        return "A.method1()";
    }
    public String method2() {
        return "A.method2() > " + this.method1();
    }
    public String method5() {
        return "A.method5() > " + this.method2();
    }
}

public class B extends A {
    public String method1() {
        return "B.method1()";
    }
    public String method3() {
        return "B.method3() > " + super.method1();
    }
    public String method4() {
        return "B.method4() > " + super.method2();
    }
    public String method5() {
        return "B.method5() > " + super.method5();
    }
}
```

```
public class C extends B {
    public String method2() {
        return "C.method2() > " + this.method1();
    }
}

public static void main(String[] args) {
    /* basic method look-up */
    System.out.println("1. " + new C().method1());
    System.out.println("2. " + new B().method1());
    System.out.println("3. " + new A().method1());

    /* this */
    System.out.println("4. " + new C().method2());
    System.out.println("5. " + new B().method2());
    System.out.println("6. " + new A().method2());

    /* super */
    System.out.println("7. " + new B().method3());
    System.out.println("8. " + new C().method4());
    System.out.println("9. " + new C().method5());
}
```


Visibility 1

```
public class A {  
    private String method1() {  
        return "A.method1()";  
    }  
    public String method2() {  
        return "A.method2() > " + this.method1();  
    }  
}
```

```
public class B extends A {  
    public String method1() {  
        return "B.method1()";  
    }  
    public static void main(String[] args) {  
        System.out.println(new B().method2());  
    }  
}
```

Visibility 1

```
public class A {  
    private String method1() {  
        return "A.method1()";  
    }  
    public String method2() {  
        return "A.method2() > " + this.method1();  
    }  
}
```

```
public class B extends A {  
    public String method1() {  
        return "B.method1()";  
    }  
    public static void main(String[] args) {  
        System.out.println(new B().method2());  
    }  
}
```

Visibility 2

```
public class C {  
    protected String method1() {  
        return "C.method1()";  
    }  
    public String method2() {  
        return "C.method2() > " + this.method1();  
    }  
}
```

```
public class D extends C {  
    public String method1() {  
        return "D.method1()";  
    }  
    public static void main(String[] args) {  
        System.out.println(new D().method2());  
    }  
}
```

Visibility 3

```
public class A {
    private String method1() {
        return "A.method1()";
    }
    public String method2() {
        return "A.method2() > " + this.method1();
    }
}

public class B extends A {
    public String method1() {
        return "B.method1()";
    }
}

public class C {
    protected String method1() {
        return "C.method1()";
    }
    public String method2() {
        return "C.method2() > " + this.method1();
    }
}

public class D extends C {
    public String method1() {
        return "D.method1()";
    }
}
```

```
public class E {
    public String method1() {
        return "E.method1()";
    }
    public String method2() {
        return "E.method2() > " + this.method1();
    }
}

public class F extends E {
    public String method1() {
        return "F.method1()";
    }
}

public class G {
    public static void main(String[] args) {
        System.out.println("1. " + new A().method2());
        System.out.println("2. " + new B().method2());
        System.out.println("3. " + new C().method2());
        System.out.println("4. " + new D().method2());
        System.out.println("5. " + new E().method2());
        System.out.println("6. " + new F().method2());
    }
}
```

Accessibility

```
public class Animal {
    private String name;

    public Animal(String name) {
        this.name = name;
    }

    private String getName() {
        return name;
    }

    public String getPair(Animal paired) {
        return this.getName() + " with " + paired.getName();
    }

    public static void main(String[] args) {
        System.out.println("1. " + new Animal("Jirafa").getPair(new Animal("Antilope")));
        System.out.println("2. " + new Animal("Tigre").getName());
    }
}
```

Accessibility

```
public class Animal {  
    private String name;  
  
    public Animal(String name) {  
        this.name = name;  
    }  
  
    private String getName() {  
        return name;  
    }  
  
    public String getPair(Animal paired) {  
        return this.getName() + " with " + paired.getName();  
    }  
  
    public static void main(String[] args) {  
        System.out.println("1. " + new Animal("Jirafa").getPair(new Animal("Antilope")));  
        System.out.println("2. " + new Animal("Tigre").getName());  
    }  
}
```

Accessibility

```
public class Animal {  
    private String name;  
  
    public Animal(String name) {  
        this.name = name;  
    }  
  
    private String getName() {  
        return name;  
    }  
  
    public String getPair(Animal paired) {  
        return this.getName() + " with " + paired.getName();  
    }  
  
    public static void main(String[] args) {  
        System.out.println("1. " + new Animal("Jirafa").getPair(new Animal("Antilope")));  
        System.out.println("2. " + new Animal("Tigre").getName());  
    }  
}
```

Overloading

```
public class A {  
    String m(A o1, B o2) {  
        return "A.m(A,B)";  
    }  
}  
  
public class B extends A {  
    String m(A o1, A o2) {  
        return "B.m(A,A)";  
    }  
  
    public static void main(String[] args) {  
        System.out.println("1. " + new B().m(new A(), new A()));  
        System.out.println("2. " + new B().m(new A(), new B()));  
  
        A object1 = new B();  
        A object2 = new B();  
  
        System.out.println("3. " + new B().m(object1, object2));  
        System.out.println("4. " + new B().m((B) object1, object2));  
        System.out.println("5. " + new B().m((B) object1, (B) object2));  
    }  
}
```


Static methods

```
public class A {
    public static String method1() {
        return "A.method1()";
    }

    public String method2() {
        return "A.method2() > " + method1();
    }
}

public class B extends A {
    public static String method1() {
        return "B.method1()";
    }

    public static void main(String[] args) {
        System.out.println("1. " + new A().method2());
        System.out.println("2. " + new B().method2());
        System.out.println("3. " + A.method1());
        System.out.println("4. " + B.method1());
        System.out.println("5. " + method1());
    }
}
```

Constructors

```
public class A {  
    public A() {  
        System.out.print("new A() > ");  
    }  
}
```

```
public class B extends A {  
    public B() {  
        System.out.print("new B() > ");  
    }  
}
```

```
public class D extends C {  
    public D() {  
        this("dog");  
        System.out.print("new D() > ");  
    }  
    public D(String name) {  
        super(name);  
        System.out.print("new D(" + name + ") > ");  
    }  
    public D(int number) {  
        super();  
        System.out.print("new D(" + number + ") > ");  
    }  
}
```

```
public class C extends A {  
    public C() {  
        System.out.print("new C() > ");  
    }  
    public C(String name) {  
        System.out.print("new C(" + name + ") > ");  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        System.out.print("1. ");  
        new A();  
        System.out.print("2. ");  
        new B();  
        System.out.print("3. ");  
        new C();  
        System.out.print("4. ");  
        new C("animal");  
        System.out.print("5. ");  
        new D();  
        System.out.print("6. ");  
        new D("jirafa");  
        System.out.print("7. ");  
        new D(3);  
    }  
}
```



Ciencias de la
Computación

FACULTAD DE CIENCIAS
FÍSICAS Y MATEMÁTICAS
UNIVERSIDAD DE CHILE

CC3002 – Metodologías de
Diseño y Programación

Auxiliar 2: Constructores y herencia

Juan-Pablo Silva
jpsilva@dcc.uchile.cl