



Victoria University
of Wellington, New Zealand
*Te Whare Wananga o te
Upoko o te Ika a Maui
Aotearoa*



SWEN221: Software Design and Engineering

8: Inheritance II


David J. Pearce & Nicholas Cameron & James Noble & Petra Malik
Computer Science, Victoria University

Inheritance + Method overriding

- Can **override** methods of superclass:

```
class A {  
    void aMethod() {  
        System.out.println("A called");  
    }  
}  
class B extends A {  
    void aMethod() {  
        System.out.println("B called");  
    }  
}  
A y = new B();  
y.aMethod();
```

B.aMethod()
overrides
A.aMethod()



Inheritance + Method overriding

- Can reuse overridden methods with **super**:

```
class A {  
    void aMethod() {  
        System.out.println("A called");  
    }  
}  
  
class B extends A {  
    void aMethod() {  
  
    }  
}
```

Inheritance + Method overriding

- Can reuse overridden methods with **super**:

```
class A {  
    void aMethod() {  
        System.out.println("A called");  
    }  
}  
  
class B extends A {  
    void aMethod() {  
        super.aMethod();  
        System.out.println("B called");  
    }  
}  
  
B y = new B();  
y.aMethod(); // prints: "A called  
              //           B called"
```

Time Travelling code!

```

class W2017 { //written in the past
    static int sum(List<Integer> l) {
        int res=0;
        for(int i=0;i<l.size();i++) {
            res+=l.get(i); //When called from W2018.main, calls a get()
        } //implementation wrote before 2017, all right!
        return res;
    } }


class W2018 { //present call the past, ok!
    public static void main(String[] arg) {
        List<Integer> l=Arrays.asList(1,2,3,4);
        System.out.println(W2017.sum(l));
    } }

class W2019 extends ArrayList<Integer>{
    @Override public Integer get(int i) { //future calls the past,
        System.out.println("Hi from 2019"); //that calls the future back!
        return super.get(i);
    }

    public static void main(String[] arg) {
        List<Integer> l=new W2019();
        l.addAll(Arrays.asList(1,2,3,4));
        System.out.println(W2017.sum(l));
    }
}

```

Time Travelling code!



```

class W2017 { //written in the past
    static int sum(List<Integer> l) {
        int res=0;
        for(int i=0;i<l.size();i++) {
            res+=l.get(i); //When called from W2019.main, calls a get()
        } //implementation wrote in 2019: time travel here!
        return res;
    }
}

class W2018 { //present call the past, ok!
    public static void main(String[] arg) {
        List<Integer> l=Arrays.asList(1,2,3,4);
        System.out.println(W2017.sum(l));
    }
}

class W2019 extends ArrayList<Integer>{
    @Override public Integer get(int i) { //future calls the past,
        System.out.println("Hi from 2019"); //that calls the future back!
        return super.get(i);
    }
    public static void main(String[] arg) {
        List<Integer> l=new W2019();
        l.addAll(Arrays.asList(1,2,3,4));
        System.out.println(W2017.sum(l));
    }
}

```

Method overloading

- Two methods can have same name!
 - Require different parameter types

```
class Car {  
    void shutDoor(Person p) {  
        System.out.println("Door shuts");  
    }  
    void shutDoor(StrongPerson s) {  
        System.out.println("Door SLAMS!");  
    }  
}
```

- Unfortunately, it is **dangerous** to do this
 - Ok, when different number of parameters

Quiz – what gets printed?

```
class Person { ... }  
class StrongPerson extends Person { ... }  
  
class Car {  
    void shutDoor(Person p) {  
        System.out.println("Door shuts");  
    }  
    void shutDoor(StrongPerson s) {  
        System.out.println("Door SLAMS!");  
    }  
}  
  
Car c = new Car();  
Person jim = new StrongPerson();  
StrongPerson henry = new StrongPerson();  
c.shutDoor(jim);  
c.shutDoor(henry);
```

A)

“Door shuts”

“Door shuts”

B)

“Door SLAMS!”

“Door SLAMS!”

C)

“Door shuts”

“Door SLAMS!”

Quiz – what gets printed?

```
class Person { ... }  
class StrongPerson extends Person { ... }  
class Car {  
    void shutDoor(Person p) {  
        System.out.println("Door shuts");  
    }  
    void shutDoor(StrongPerson s) {  
        System.out.println("Door SLAMS!");  
    }  
}  
  
Car c = new Car();  
Person jim = new StrongPerson();  
StrongPerson henry = new StrongPerson();  
c.shutDoor(jim);  
c.shutDoor(henry);
```

A)

~~"Door shuts"~~
~~"Door shuts"~~

B)

~~"Door SLAMS!"~~
~~"Door SLAMS!"~~

C)

✓ "Door shuts"
✓ "Door SLAMS!"

Quiz – what gets printed?

```
class Person { ... }  
class StrongPerson extends Person { ... }  
class Car {  
    void shutDoor(Person p) {  
        System.out.println("Door shuts");  
    }  
    void shutDoor(StrongPerson s) {  
        System.out.println("Door SLAMS!");  
    }  
}  
  
Car c = new Car();  
Person jim = new StrongPerson();  
Person henry = new StrongPerson();  
c.shutDoor(jim);  
c.shutDoor(henry);
```

A)

“Door shuts”

“Door shuts”

B)

“Door SLAMS!”

“Door SLAMS!”

C)

“Door shuts”

“Door SLAMS!”

Quiz – what gets printed?

```
class Person { ... }  
class StrongPerson extends Person { ... }  
class Car {  
    void shutDoor(Person p) {  
        System.out.println("Door shuts");  
    }  
    void shutDoor(StrongPerson s) {  
        System.out.println("Door SLAMS!");  
    }  
}  
  
Car c = new Car();  
Person jim = new StrongPerson();  
Person henry = new StrongPerson();  
c.shutDoor(jim);  
c.shutDoor(henry);
```

A)

“Door shuts”

“Door shuts”

B)

“Door SLAMS!”

“Door SLAMS!”

C)

“Door shuts”

“Door SLAMS!”

Quiz – what gets printed?

```
class Pow {  
    public int pow(int base,int exp) {  
        if(exp==0) return 1;  
        return base*pow(base,exp-1);  
    }  
}
```

```
class PowLog extends Pow {  
    public int pow(int base,int exp) {  
        System.out.println("LogMessage");  
        return super.pow(base,exp);  
    }  
}
```

```
System.out.println(new PowLog().pow(4,3));
```

A) **LogMessage**
64

LogMessage

LogMessage

B) **LogMessage**
64

Quiz – what gets printed?

```
class Pow {  
    public int pow(int base,int exp) {  
        if(exp==0) return 1;  
        return base*pow(base,exp-1);  
    }  
}
```

```
class PowLog extends Pow {  
    public int pow(int base,int exp) {  
        System.out.println("LogMessage");  
        return super.pow(base,exp);  
    }  
}
```

```
System.out.println(new PowLog().pow(4,3));
```

A) LogMessage
64



LogMessage

LogMessage

B) LogMessage
64



Inheritance and Code Reuse

```
class A {  
    private int value;  
    public int add(int x) {  
        return value+x;  
    }  
    ... // other operations  
}  
  
class B {  
    private int value;  
    public int add(int x) {  
        return value+x;  
    }  
    ... // other operations  
}
```

Inheritance and Code Reuse

```
class A {  
    private int value;  
    public int add(int x) {  
        return value+x;  
    }  
    ... // other operations  
}  
  
class B {  
    private int value;  
    public int add(int x) {  
        return value+x;  
    }  
    ... // other operations  
}
```



```
class C {  
    private int value;  
    public int add(int x) {  
        return value+x;  
    }  
}  
  
class A extends C {  
    ... // other operations  
}  
  
class B extends C {  
    ... // other operations  
}
```

Protected Members

```
class A {  
    private int value;  
    public int add(int x) {  
        return value+x;  
    }  
    int otherOp() {  
        return value+1;  
    }  
}
```



```
class C {  
    private int value;  
    public int add(int x) {  
        return value+x;  
    }  
}  
  
class A extends C {  
    int otherOp() {  
        return value+1;  
    }  
}
```

- Version on right no longer compiles!
 - Because value is **private** to C

Protected Members

```
class A {  
    private int value;  
    public int add(int x) {  
        return value+x;  
    }  
    int otheOp() {  
        return value+1;  
    }  
}
```



```
class C {  
    protected int value;  
    public int add(int x) {  
        return value+x;  
    }  
}  
  
class A extends C {  
    int otheOp() {  
        return value+1;  
    }  
}
```

- Now it compiles (but, is still not good)
 - Because value is **protected** in C
 - **Beware!!** This approach should be avoided: here C exposes its concrete fields in its (protected) interface

Protected Members

```
class A {  
    private int value;  
    public int add(int x) {  
        return value+x;  
    }  
    int otherOp() {  
        return value+1;  
    }  
}
```



```
class C {  
    private int value;  
    public int add(int x) {  
        return value+x;  
    }  
    protected int value() {  
        return value;  
    }  
}  
class A extends C {  
    int otherOp() {  
        return value()+1;  
    }  
}
```

- Ok, now it's good!
 - Because value is **private** in C, but still accessible
 - Note how we can chose to not offer the setter

Abstract Classes and Interfaces

- From Java8, both Abstract classes and interfaces can be used for code reuse.
- They both can contain abstract methods as well as implemented/default methods.
- Abstract methods:
 - Have no implementation
 - Concrete subclasses must provide it
- Both can not be instantiated
- Abstract classes may also contain fields and constructors
 - Interfaces can have abstract getters and setters to “imagine” the presence of fields.
- When field+constructors are not strictly needed, interfaces should be used instead of abstract classes

Abstract Classes and Interfaces

All concrete subclasses of length must have metres() and yards() methods


Code reuse possible after Java8

```
interface Length {  
    double metres();  
    double yards();  
    default Length add(Length l) {  
        return new Yards(l.yards() + yards());  
    }  
}  
  
class Yards implements Length {  
    private int yards;  
    public double metres() { return yards*0.91 ; }  
    public double yards() { return yards; }  
}  
  
class Metres implements Length {  
    private int metres;  
    public double metres() { return metres; }  
    public double yards() {return metres*1.09; }  
}
```

Interfaces

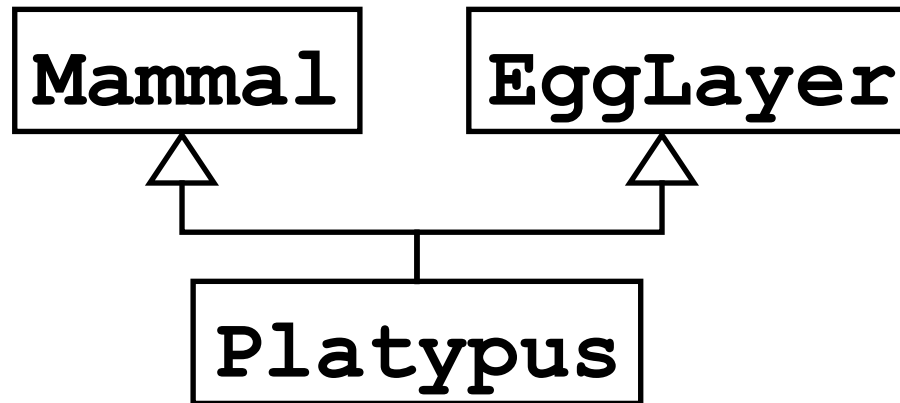
- Separate interface from implementation
 - Implementation can change without breaking system
 - Interfaces declare what operations must be supported
 - Interface poses no additional constraint on implementation!
 - Classes then implement the interface

```
public interface Length {  
    double metres();  
    double yards();  
}
```



All implementations of Length must have metres() and yards() methods

Multiple Inheritance



- In Java, this is not possible!
 - A class cannot have more than one superclass
 - Other languages (e.g. C++) support this
 - But, a class can implement **more than one interface**

```
class Platypus extends Mammal, EggLayer { ... }
```

```
class Platypus implements Mammal, EggLayer { ... }
```

Inheritance + constructors

- Super can/must be used in a constructor to reuse the superclass constructor:

```
class A {  
    A(Object aParam) {...}  
}  
  
class B extends A {  
    B(Object aParam, Object anotherParam) {  
        super(aParam);  
        ...  
    }  
}
```

Inheritance + constructors

- “**this**” call is similar to the super call, and can be used in a constructor to reuse the another constructor in the same class:

```
class A {  
    A(Object aParam) {...}  
    A() { this("Hi"); }  
}
```


Inheritance + Final classes

- Final classes cannot be extended!
- Final methods can not be overridden!

```
final class A {  
    ...  
}  
class B extends A { // ERROR  
    ...  
}
```

How to use:

- (final)classes, interfaces, abstract classes
- (final)methods,fields and constructors
- visibility, overriding, overloading

- So that code is easy to:
 - understand
 - modify
 - test

How to use:

- We are **changing our mind all the time**.
- A drastic proposal originally from Google gurus:
 - Everything is either public or private
 - All classes should be final or abstract
 - All public methods should be either final or abstract (or static)
 - All types (method parameters and return types) should be interfaces.
 - All fields and constructors should be private