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

SWEN221: Software Design and Engineering

6: Inheritance II

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

Inheritance + Method overriding

- Can reuse overriden 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"
```

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!”

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 otheOp() {
  return value+1;
}}
```

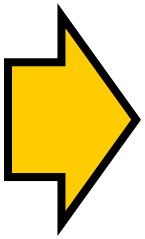
```
class C {
 protected int value;
 public int add(int x) {
  return value+x;
 }
}

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 as it results in the *fragile base-class problem*

Protected Members

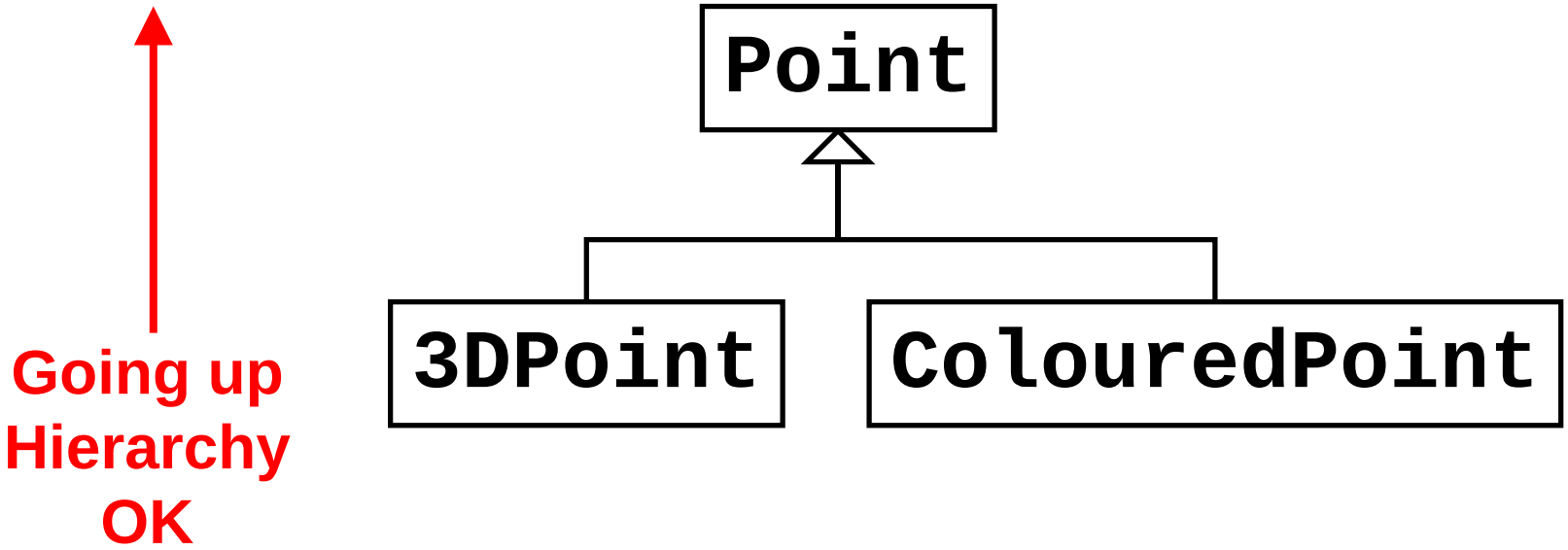
```
class A {
 private int value;
 public int add(int x) {
  return value+x;
 }
 int otherOp() {
  return value+1;
}}
```



```
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
  - The *fragile base-class problem* will be discussed in SWEN222

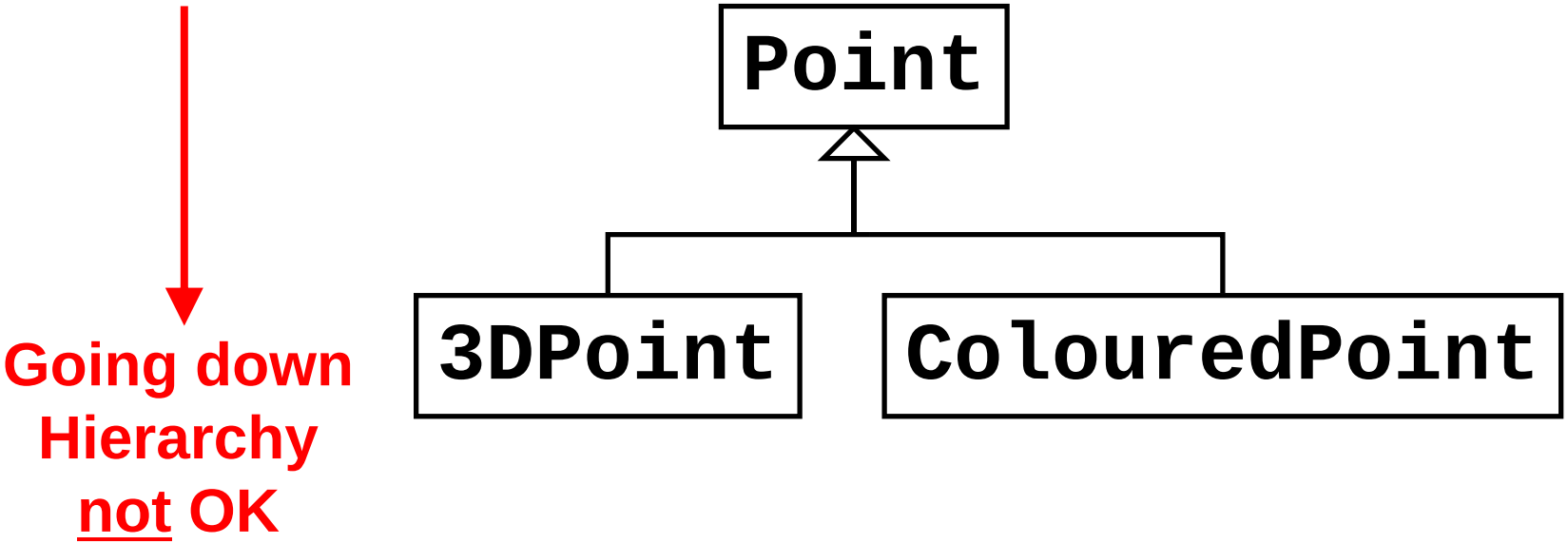
Up Casting




```
3DPoint doSomething() { … }

Point p = doSomething();
```

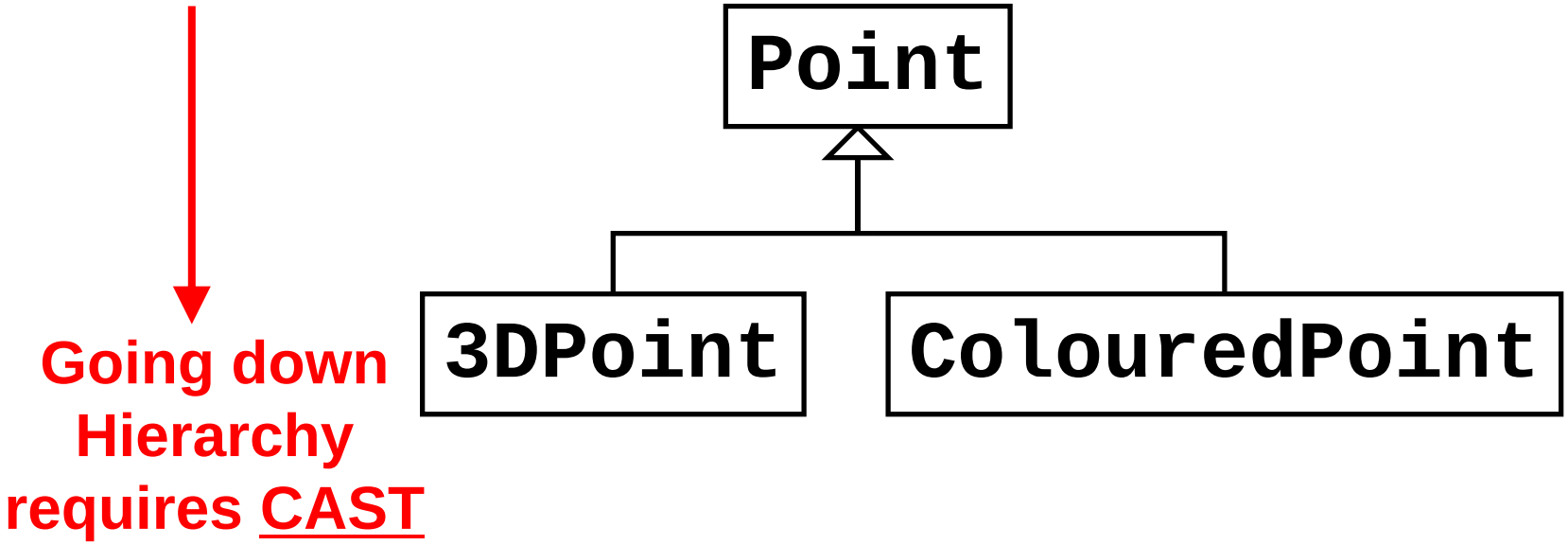
Down Casting




```
Point doSomething() { … }

3DPoint p = doSomething();
```

Down Casting




```
Point doSomething() { … }

3DPoint p = (3DPoint) doSomething();
```

- Will throw exception if not 3DPoint!

Instanceof

- Can use **`instanceof`** to check whether subtype or not:

```
Point p = …
if(p instanceof 3DPoint) {
 3DPoint dp = (3DPoint) p;
 …
} else {
 …
}
```

- But you probably shouldn't!

Abstract Classes

- Abstract classes:
  - Contain **abstract methods**
  - May also contain concrete methods + fields
  - **Cannot be instantiated**
  - Similar to interfaces in some ways

- Abstract methods:
  - Have no implementation
  - Concrete subclasses <u>must</u> provide it

Abstract Classes

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

Code reuse not possible with interface

```
abstract class Length {
 abstract double metres();
 abstract double yards();
 public Length add(Length l) {
  return new Yards(l.yards() + yards());
}}
class Yards extends Length {
 private int yards;
 public double metres() { return yards*0.91 ; }
 public double yards() { return yards; }
}
class Metres extends 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
  - Classes then implement the interface

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

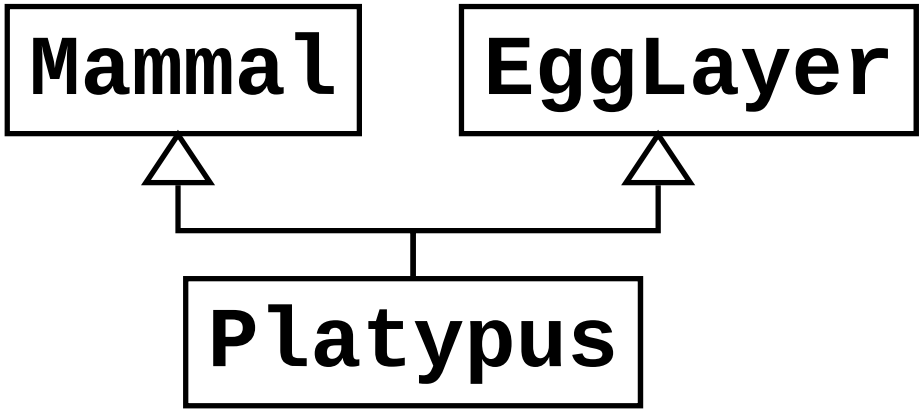
All implementations of Length must have metres() and yards() methods

Example using interfaces

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

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 { … }
```

Inhertiance + constructors

- Super can also 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 + Final classes

- Final classes cannot be extended!

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