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

SWEN221 Software Development

Inheritance II

Thomas Kuehne

Victoria University

(slides modified from slides by David J. Pearce & Nicholas Cameron & James Noble & Petra Malik)

Method overloading

- Two methods can have same name
  - as long as the parameter signature differs

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

Method overloading

- be sure not to let overloading interfere with overriding

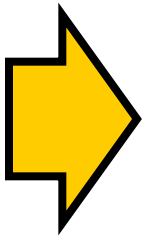
```
class Car {
 void shutDoor(Person p) {
  System.out.println(“Door shuts”);
 }}

class BigCar {
 void shutDoor(StrongPerson s) {
  System.out.println(“Door SLAMS!”);
}}
```

- here the type of the argument (instead of the receiver) **determines** the **method selection**

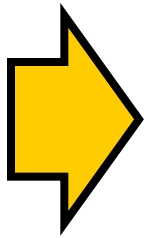
Inheritance and Code Reuse

```
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;
 }
 … // other operations
}
```



Protected Members

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



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

class B extends A {
 int otheOp() {
  return value+1;
}}
```

- Now it compiles
  - variable is now **protected** in A
  - still not ideal as it results in the *fragile base-class* problem (→ SWEN 222)

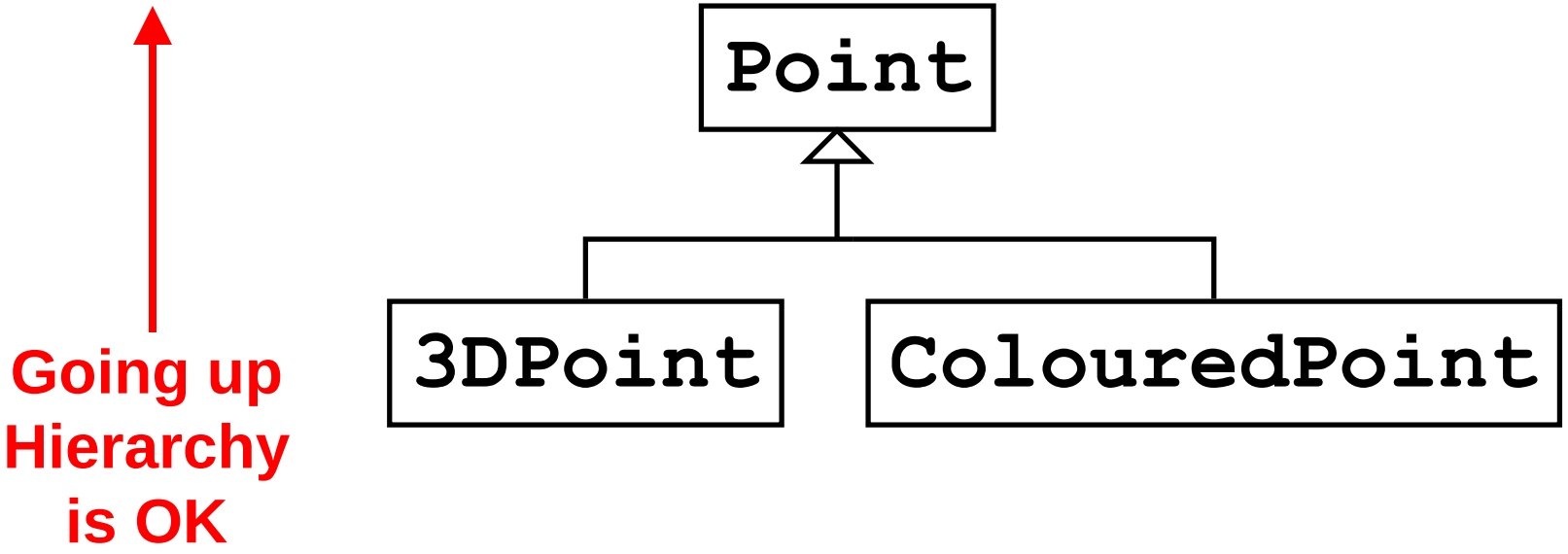
Protected Members

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

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

- Better
  - because value is **private** in C, but still accessible
  - B does not depend directly on implementation choices of A anymore

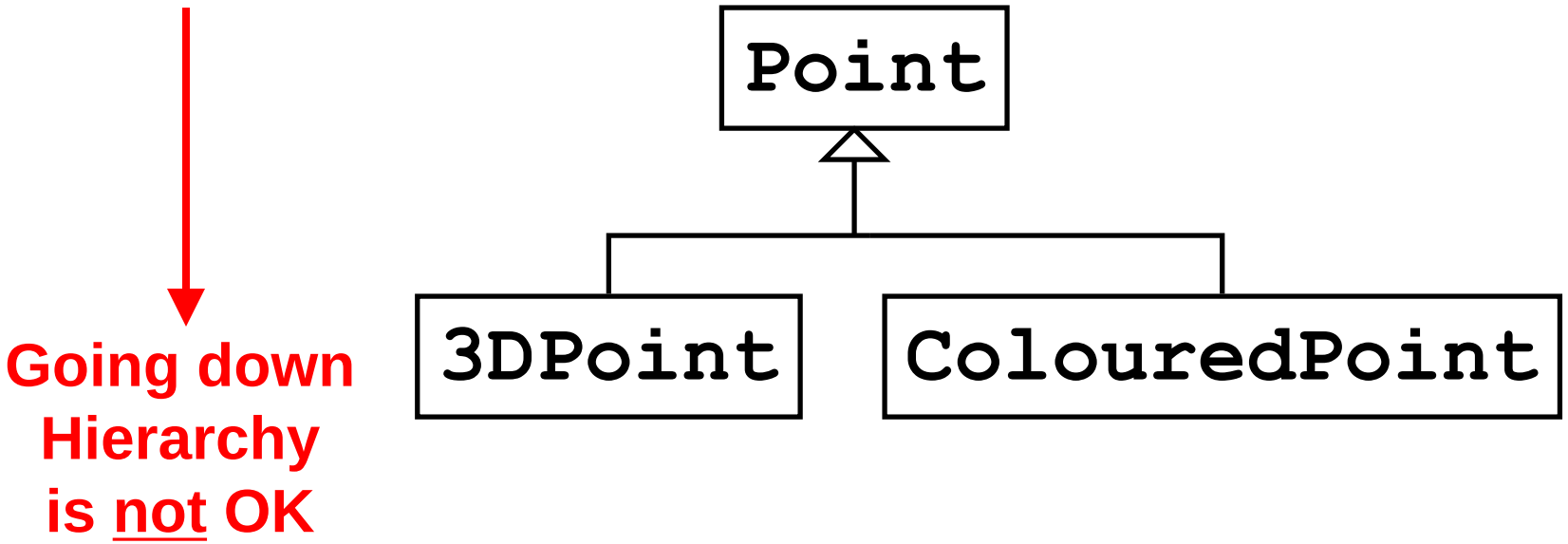
Assignment Compatibility




```
3DPoint doSomething() { … }

Point p = doSomething();
```

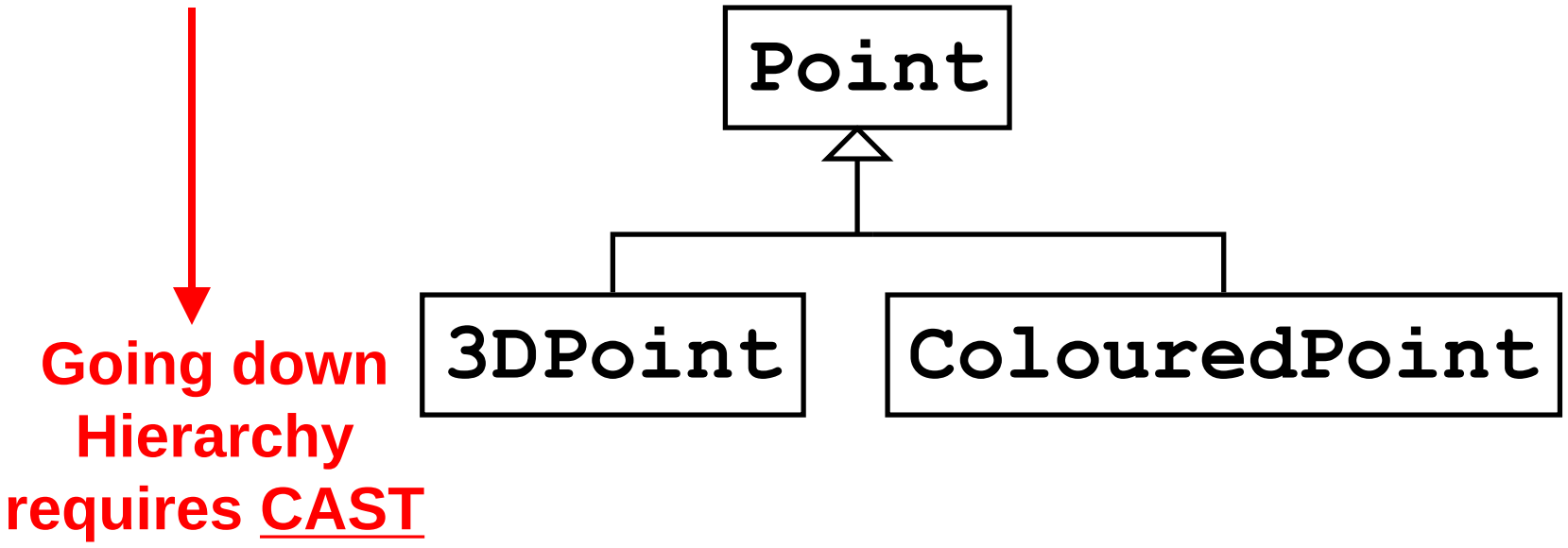
Assignment Compatibility




```
Point doSomething() { … }

3DPoint p = doSomething();
```

Down Casting




```
Point doSomething() { … }

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

- Will throw exception if cast cannot be performed

Instanceof

- Can use **`instanceof`** to check whether cast attempt will succeed

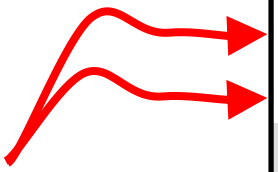
```
Point p = …
if(p instanceof 3DPoint) {
 3DPoint dp = (3DPoint) p;
 ...
} else {
 ...
}
```

- Avoid explicit dependence on object types wherever possible, though

Abstract Classes

- Abstract classes:
  - Contain **abstract methods**
  - May also contain concrete methods + fields
  - **Cannot be instantiated**
  - Similar to interfaces in particular since interfaces gained the ability to have default implementations
- Abstract methods:
  - Have no implementation
  - **Concrete** subclasses **must** provide it

Abstract Classes



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

```
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*0.9144 ; }
 public double yards() { return yards; }
}
class Metres extends Length {
 private int metres;
 public double metres() { return metres; }
 public double yards() {return metres*1.093613; }
}
```

Interfaces

- Separate interface from implementation
  - Interfaces declare what operations must be supported
  - Classes then implement the interface
  - Implementation can change without breaking system

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

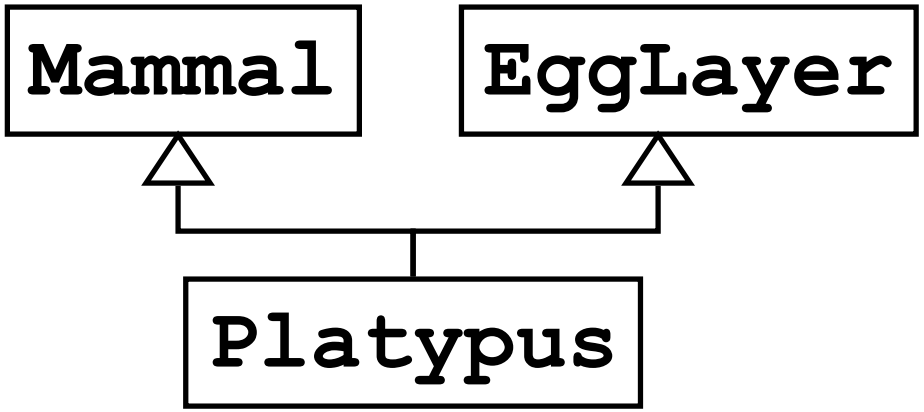
All implementations of **Length** must have **metres()** and **yards()** methods

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++, Eiffel) support this
  - But, a class can implement **more than one interface**

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

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

Inheritance + Final classes

- Final classes cannot be extended!

```
final class A {
 …
}

class B extends A { // ERROR
 …
}
```