



UNIVERSITY OF
ABERDEEN

JC2002 Java Programming

Lecture 10: Composition and polymorphism

Class relationships

- Inheritance relationship is basically *is-a* relationship
 - Car (subclass) *is a* vehicle (superclass)
 - Dog (subclass) *is a* mammal (superclass)
- However, some class relationships are *has-a* relationships
 - Car *has* an engine
 - Dog *has* a tail
 - Person *has* a name
 - Has-a relationships should be created by *composition* of existing classes, rather than inheritance

Composition

- A class can have references to objects of other classes as members
 - This is called composition and is sometimes referred to as has-a relationship
- Composition is used to ease complexity, which lets us create objects with fewer dependencies
 - Example: An AlarmClock object needs to know the current time and the time when it is supposed to sound its alarm, so it is reasonable to include two references to Time objects in an AlarmClock object

Composition example

Car.java

```
1 public class Car {  
2     private Engine engine;  
3     public Car() {  
4         this.engine = new Engine();  
5     }  
6     public void startCar() {  
7         engine.makeNoise();  
8     }  
9 }
```

Car has an Engine

Engine.java

```
1 public class Engine {  
2     public void makeNoise() {  
3         System.out.println("Wrroom!");  
4     }  
5 }
```

Composition or inheritance?

- There has been much discussion in the software engineering community about the relative merits of composition and inheritance
 - Each has its own place, but inheritance is often overused and composition is more appropriate in many cases
- A mix of composition and inheritance often is the best approach
 - It is best to think whether *is-a* or *has-a* relationship represents your case more naturally

Composition vs. inheritance

Composition

- Composition and aggregation form has-a relationships where sum is greater than its parts
- Objects stand alone, so development cost is higher: fewer built-in dependencies that can be reused

Inheritance

- Inheritance for when message delegation is free within hierarchy
- Easier to develop, but more dependencies: it is easy to break things by changing something in a superclass that affects all the subclasses

Nested classes

- Java allows declaring classes inside classes (nested classes)
 - To instantiate a nested (inner) class, you need to first instantiate the enclosing (outer) class
 - Non-static inner classes have access to other members of the outer class, even if declared **private**
- Nested classes can be considered as a kind of “composition”, since the outer class “owns” the inner class
 - However, some benefits of composition are lost, such as polymorphic behavior and reusability: only use a nested class, if you are absolutely sure that you do not need it anywhere else!

Nested class example

Car.java

```
1  public class Car {  
2      private class Engine {  
3          public void makeNoise() {  
4              System.out.println("Wrroom!");  
5          }  
6      }  
7      private Engine engine;  
8      public Car() {  
9          this.engine = new Engine();  
10     }  
11     public void startCar() {  
12         engine.makeNoise();  
13     }  
14     public static void main(String[] args) {  
15         Car car = new Car();  
16         car.startCar();  
17     }  
18 }
```



Nested class defined here

```
$ java Car  
Wrroom!  
$
```

Anonymous classes

- In Java, you can declare anonymous classes
 - Anonymous classes are like local classes, except that they do not have a name
 - Use them if you only need to use a local class in one place
- Anonymous classes are defined in their initialisation statements when they are instantiated
 - Declare anonymous classes using the following syntax:

```
SuperClass myClass = new SuperClass() {  
    // override methods here as needed  
};
```

Anonymous class example

Car.java

```
1  class Engine {  
2      public void makeNoise() {  
3          System.out.println("Put put put!");  
4      }  
5  }  
6  public class Car {  
7      private Engine engine;  
8      public Car() {  
9          this.engine = new Engine() {  
10             public void makeNoise() {  
11                 System.out.println("Wrrooom!");  
12             }  
13         };  
14     }  
15 }
```

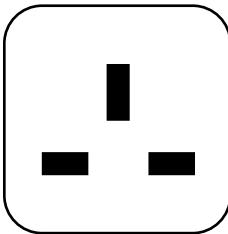
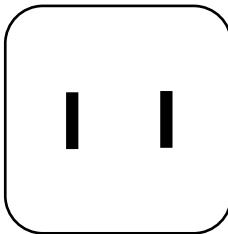
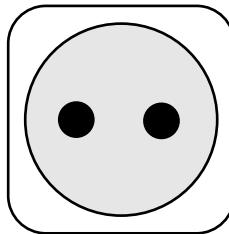
```
15  public void startCar() {  
16      engine.makeNoise();  
17  }  
18  public static void main(String[] args) {  
19      Car car = new Car();  
20      car.startCar();  
21  }  
22 }
```

```
$ java Car  
Wrrooom!  
$
```

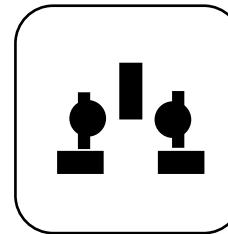
Anonymous subclass of
Engine defined here

Polymorphism

- *Polymorphism* allows you to define one interface and have multiple implementations
 - The word “poly” means many and “morphs” means forms: polymorphism means “many forms”



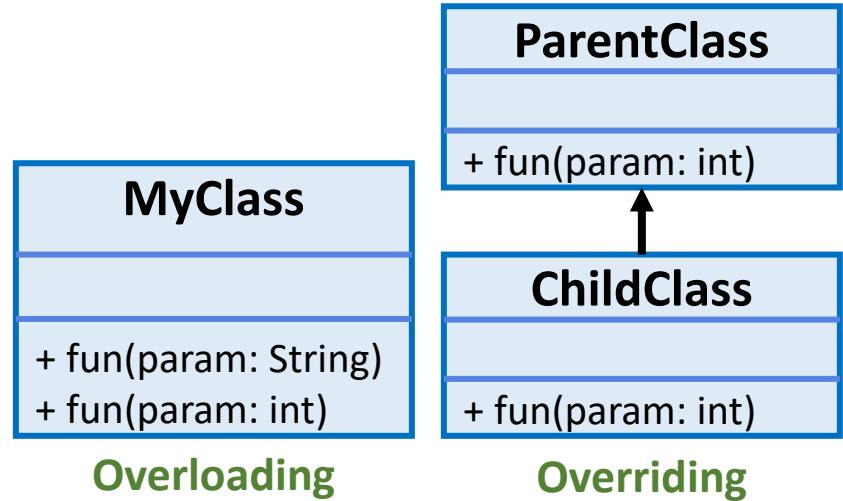
Without polymorphism



With polymorphism

Method overloading and overriding

- In Java, polymorphism is mainly divided into two types:
 - Compile-time polymorphism (static polymorphism achieved by *method overloading*)
 - Runtime polymorphism (dynamic method dispatch achieved by *method overriding*)



Overloading example

- We discuss overloading of constructors already, but other methods can be overloaded as well

```
1 class Helper {  
2     static int Multiply(int a, int b) {return a * b;}  
3     static double Multiply(double a, double b) {return a * b;}  
4     public static void main(String[] args)  
5     {  
6         System.out.println(Helper.Multiply(2, 4));  
7         System.out.println(Helper.Multiply(4.2, 3.8));  
8     }  
9 }
```

```
$ java Helper  
8  
15.540000000000001
```

Overloading example (2)

- Different versions of the method can differ in parameter types or the number of parameters

```
1 class Helper {  
2     static int Multiply(int a, int b) {return a * b;}  
3     static int Multiply(int a, int b, int c) {return a * b * c;}  
4     public static void main(String[] args)  
5     {  
6         System.out.println(Helper.Multiply(2, 4));  
7         System.out.println(Helper.Multiply(2, 4, 8));  
8     }  
9 }
```



```
$ java Helper  
8  
64
```

Runtime overriding example

```
1  class Vehicle {  
2      public void printType() {  
3          System.out.println("undefined");  
4      }  
5  }  
6  class Car extends Vehicle {  
7      public void printType() {  
8          System.out.println("car");  
9      }  
10 }  
11 class MotorBike extends Vehicle {  
12     public void printType() {  
13         System.out.println("motorbike");  
14     }  
15 }
```

```
16  public class TestEngines {  
17      public static void main(String[] arg) {  
18          Vehicle vehicle = new MotorBike();  
19          System.out.print("Vehicle type 1: ");  
20          vehicle.printType();  
21          vehicle = new Car();  
22          System.out.print("Vehicle type 2: ");  
23          vehicle.printType();  
24      }  
25  }
```

```
$ java TestEngines  
Vehicle type 1: motorbike  
Vehicle type 2: car  
$
```

Overriding data members

- Note that overriding works for methods but not data members!
 - Runtime polymorphism cannot be achieved by inherited variables

```
1  class Vehicle {  
2      int maxSpeed = 50;  
3  }  
4  class Car extends Vehicle {  
5      int maxSpeed = 150;  
6  }  
7  public class TestEngines {  
8      public static void main(String[] arg) {  
9          Car ford = new Car();  
10         System.out.printf("Max speed: %d%n", ford.maxSpeed);  
11     }  
12 }
```

```
$ javac TestEngines.java  
Max speed: 50  
$
```

Summary

- Java is an *object oriented language*; therefore, to understand Java, it is essential to understand the OOP concepts of Java
 - **Abstraction:** *classes, objects, methods* and *variables* provide simple representations of complex underlying data and behavior
 - **Encapsulation:** access to private members of a class can be controlled via *access modifiers*
 - **Inheritance:** inherited *subclasses* can be declared to share the attributes of the higher level *superclasses*
 - **Polymorphism:** allows methods with the same name to work in different contexts

Questions, comments?