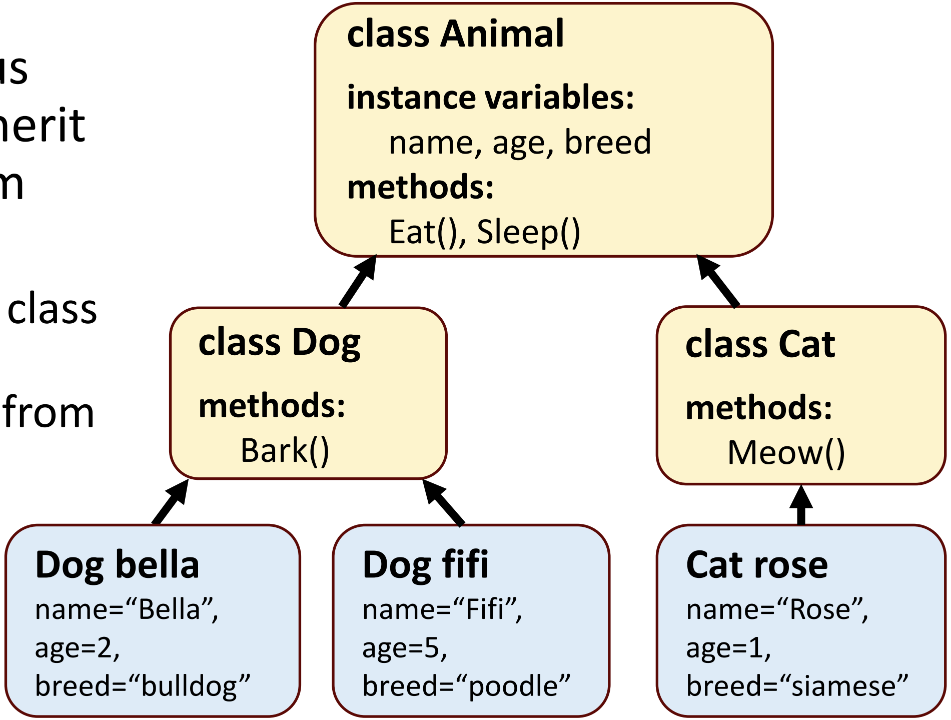
1495 UNIVERSITY OF ABERDEEN

JC2002 Java Programming

Lecture 9: Class inheritance and access modifiers

Class inheritance

- ***Class inheritance*** lets us declare classes that inherit common structure from higher level classes
  - Objects of an inherited class can use the member variables and methods from the class it inherits



class Animal

instance variables:
name, age, breed

methods:
Eat(), Sleep()

class Dog

methods:
Bark()

class Cat

methods:
Meow()

Dog bella
name="Bella",
age=2,
breed="bulldog"

Dog fifi
name="Fifi",
age=5,
breed="poodle"

Cat rose
name="Rose",
age=1,
breed="siamese"

Benefits of class inheritance

- DRY: don’t repeat yourself
  - Inheritance lets us pass on common structure and messages to similar objects
- Class inheritance allows “reuse” parts of objects
  - We can pull out common attributes and move them up to higher level object, and then differentiate them at the lower level
  - Reduces repetition and eases code maintenance and reusability

Superclasses and subclasses

- The class that inherits from another class is ***subclass*** (child)
  - Java does not support multiple inheritance directly: you can only inherit from one class
- The class being inherited from is ***superclass*** (parent)
  - Objects of all classes that extend a common superclass can be treated as objects/members of that superclass
- To inherit from a class, use **`extends`** keyword, for example:

```
class Dog extends Animal { … }
```

Inheritance example

Vehicle.java

```java
class Vehicle {
  protected String brand = "Ford";
  public void honk() {
    System.out.println("Tuut tuut!");
  }
}
```

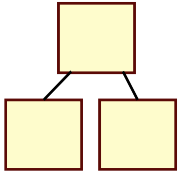
Car.java

```java
class Car extends Vehicle {
  private String modelName = "Mustang";
  public static void main(String[] args) {
    Car myCar = new Car();
    myCar.honk();
    System.out.println(myCar.brand + " " + myCar.model);
  }
}
```

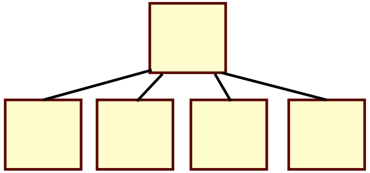
```
$ javac Car.java
$ java Car
Tuut tuut!
Ford Mustang
$
```

Inheritance hierarchies

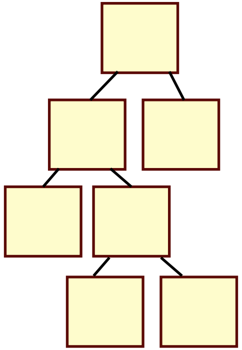
- Different class hierarchies can be constructed via inheritance
  - Deep hierarchies are complicated and tend to get wider over time, making them harder to maintain and use
  - For simplicity, shallow hierarchies are more recommended



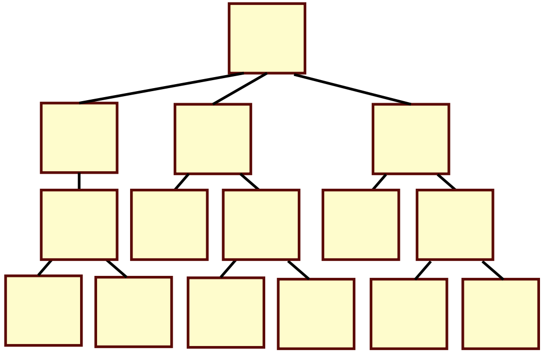
Shallow, Narrow



Shallow, Wide



Deep, Narrow



Deep, Wide

Using constructors with subclasses

- The first task of a subclass constructor is to call its direct superclass's constructor *explicitly* or *implicitly*
  - Ensures that the instance variables inherited from the superclass are initialized properly.
- If the code does not include an explicit call to the superclass's constructor, Java implicitly calls the superclass's default or no-argument constructor

Constructor example

TestCar.java

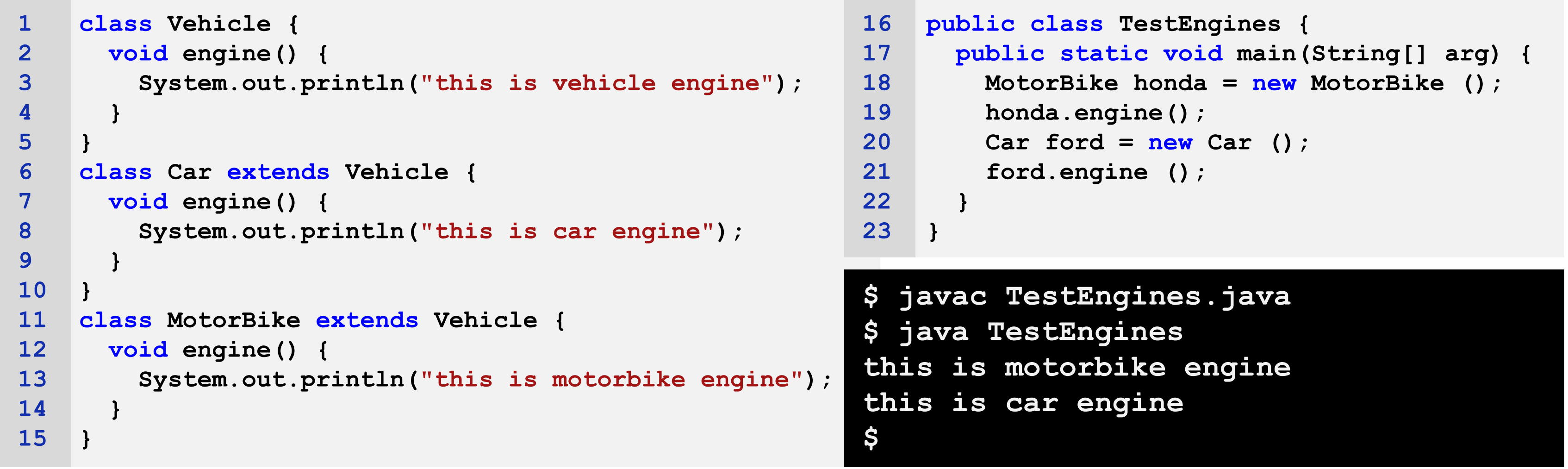
```java
class Vehicle {
  public Vehicle() {
    System.out.println("this is Vehicle constructor");
  }
}
class Car extends Vehicle {
  public Car() {
    System.out.println("this is Car constructor");
  }
}
public class TestCar {
  public static void main(String[] arg) {
    Car ford = new Car();
  }
}
```

```
$ javac TestCar.java
$ java TestCar
this is Vehicle constructor
this is Car constructor
$
```

Redefine (override) methods

- Even when a superclass method is appropriate for a subclass, that subclass often needs a customized version of the method
- The subclass can *override* (i.e., redefine) the superclass method with an appropriate implementation
  - In Java, you can use optional **`@Override`** annotation to tell the compiler that the method is supposed to override another method; this can help to find errors during compilation time
- If keyword **`final`** is used for a method, it cannot be overridden; an attempt to override a `final` method gives a compilation error

Overriding example



```java
class Vehicle {
  void engine() {
    System.out.println("this is vehicle engine");
  }
}
class Car extends Vehicle {
  void engine() {
    System.out.println("this is car engine");
  }
}
class MotorBike extends Vehicle {
  void engine() {
    System.out.println("this is motorbike engine");
  }
}
```

```java
public class TestEngines {
  public static void main(String[] arg) {
    MotorBike honda = new MotorBike ();
    honda.engine();
    Car ford = new Car ();
    ford.engine ();
  }
}
```

```
$ javac TestEngines.java
$ java TestEngines
this is motorbike engine
this is car engine
$
```

Overriding example with @Override

@Override annotation reveals a typing error in the method name

```java
    System.out.println("this is vehicle engine");
  }
}
class Car extends Vehicle {
  @Override
  void engne()  {
    System.out.println("this is car engine");
  }
}
class MotorBike extends Vehicle {
  @Override
  void engine()  {
    System.out.println("this is motorbike engine");
  }
}
```

```java
public class TestEngines {
  public static void main(String[] arg) {
    MotorBike honda = new MotorBike ();
    honda.engine();
    Car ford = new Car ();
    ford.engine ();
  }
}
```

```
$ javac TestEngines.java
error: method does not override or
implement a method from a supertype
  @Override
  ^
1 error
$
```

Overriding example with final

```java
class Vehicle {
  final void engine() {
    System.out.println("this is vehicle engine");
  }
}
class Car extends Veh
  @Override
  void engine() {
    System.out.println("this is car engine");
  }
}
class MotorBike extends Vehicle {
  @Override
  void engine() {
    System.out.println("this is motorbike engine");
  }
}
```

```java
public class TestEngines {
  public static void main(String[] arg) {
    MotorBike honda = new MotorBike ();
    honda.engine();
    Car ford = new Car ();
    ford.engine ();
```

Method defined as final cannot be overriden

```
$ javac TestEngines.java
error: engine() in Car cannot override
engine() in Vehicle
  void engine() {
       ^
  overridden method is final
1 error
$
```

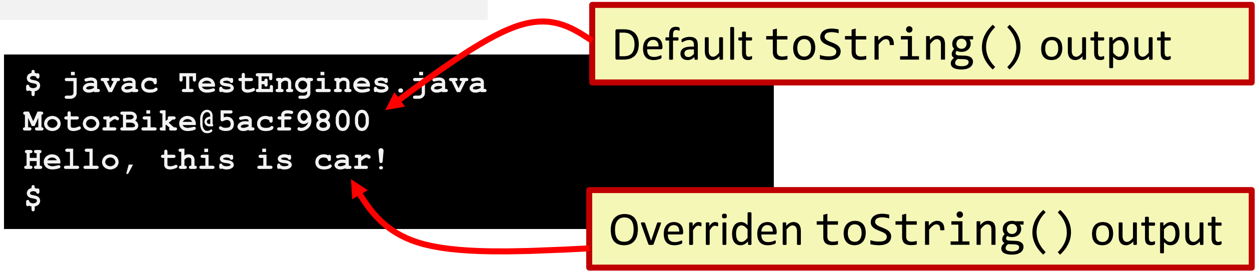
Method inheritance

- In Java, *every class* is a subclass of **`class Object`**, even if not explicitly defined to extend `Object`
- Some methods, such as `toString`, are inherited from `Object` and therefore defined for every class
    - Called implicitly whenever an object must be converted to a string representation
    - The default `toString` method returns a `String` with the name of the object's class
    - More appropriate `String` representation can be specified by overriding `toString`

Overriding example of toString() method

```java
class Vehicle {
}
class Car extends Vehicle {
  @Override
  public String toString() {
    return "Hello, this is car!";
  }
}
class MotorBike extends Vehicle {
}
```

```java
public class TestEngines {
  public static void main(String[] arg) {
    MotorBike honda = new MotorBike ();
    Car ford = new Car();
    System.out.println(honda.toString());
    System.out.println(ford.toString());
  }
}
```



```
$ javac TestEngines.java
MotorBike@5acf9800
Hello, this is car!
$
```

Default toString() output

Overriden toString() output

Access modifiers

- A class's *public* members are accessible wherever the program has a reference to an object of that class *or one of its subclasses*
- A class's *private* members are accessible only within the class itself
- To enable a subclass to directly access superclass instance variable, we can declare those members as *protected* in the superclass
  - Protected access is an intermediate level of access between public and private
  - All public and protected superclass members retain their original access modifier when they become members of the subclass

Access modifier protected

- A superclass’s protected members can be accessed by members of *that superclass*, its *subclasses*, and *other classes in the same package* (protected members also have package access)
  - Subclass methods can refer to public and protected members inherited from the superclass simply by using the member names
- Superclass’s private members are hidden from its subclasses
  - They can be accessed only through the public or protected methods inherited from the superclass
  - In many cases, it is better to use private instance variables to encourage proper software engineering

Disadvantages of protected variables

- With protected instance variables, we may need to modify all the subclasses of a superclass if the superclass implementation changes
  - Such a class is said to be fragile or brittle, because a small change in the superclass can “break” subclass implementation
  - You should be able to change the superclass implementation while still providing the same services to the subclasses
- A class’s protected members are visible to all classes in the same package as the class containing the protected members – this is not always desirable (the principle of minimum privilege)

Summary of access modifiers

Access to	default	private	protected	public
Same class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

- Access modifiers allow *encapsulation* (data hiding from other classes), one of the fundamental concepts of OOP

Calling superclass constructor

- Each subclass constructor must implicitly or explicitly call one of its superclass's constructors to initialize the instance variables inherited from the superclass
  - The syntax for calling superclass constructor: **`super(arguments)`**
  - Must be the first statement in the constructor's body
  - This lets you specify how to instantiate the object
- If the subclass constructor did not invoke the superclass's constructor explicitly, the compiler would attempt to insert a call to the superclass's default or no-argument constructor
  - You can also explicitly use `super()` to call the superclass's no-argument or default constructor, but this is not usually done

Superclass constructor example

```java
class Vehicle {
  private String type;
  public Vehicle() {
    this.type = "undefined";
  }
  public Vehicle(String type) {
    this.type = type;
  }
}
class Car extends Vehicle {
  private Engine engine;
  public Car() {
    super("car");
  }
}
```

```java
public class TestEngines {
  public static void main(String[] arg) {
    Car ford = new Car();
    System.out.print("Type: ");
    ford.printType();
  }
}
```

```
$ javac TestEngines.java
Type: car
$
```

Invokes superclass's constructor with a parameter. Note that variable **type** is private, so it cannot be accessed directly outside the superclass **Vehicle**.

Reference super methods

- When a subclass method overrides an inherited superclass method, the superclass version of the method can be accessed from the subclass by preceding the superclass method name with keyword **super** and dot(**.**) separator

```java
class Vehicle {
  public void engine() {
    System.out.println("this is vehicle engine");
  }
}
class Car extends Vehicle {
  public void engine() {
    super.engine();
    System.out.println("this is car engine");
  }
}
```

```java
public class TestEngines {
  public static void main(String[] arg) {
    Car ford = new Car ();
    ford.engine();
  }
}
```

```
$ java TestEngines
this is vehicle engine
this is car engine
$
```

Questions, comments?

1495
UNIVERSITY OF
ABERDEEN