Java Programming 2
Lecture #7

Mary Ellen Foster
MaryEllen.Foster@glasgow.ac.uk
14 October 2019

Image by Milo McDowell from http://unsplash.com/

Outline

Polymorphism

Constructors and inheritance

Method overloading

Inheritance and static methods

Polymorphism

Literal meaning: “many forms”

In OO design: whenever an instance of class A is expected, you can also use an instance of any subclass of A

If a method is overridden in a subclass, Java will always use the most-specific overridden version

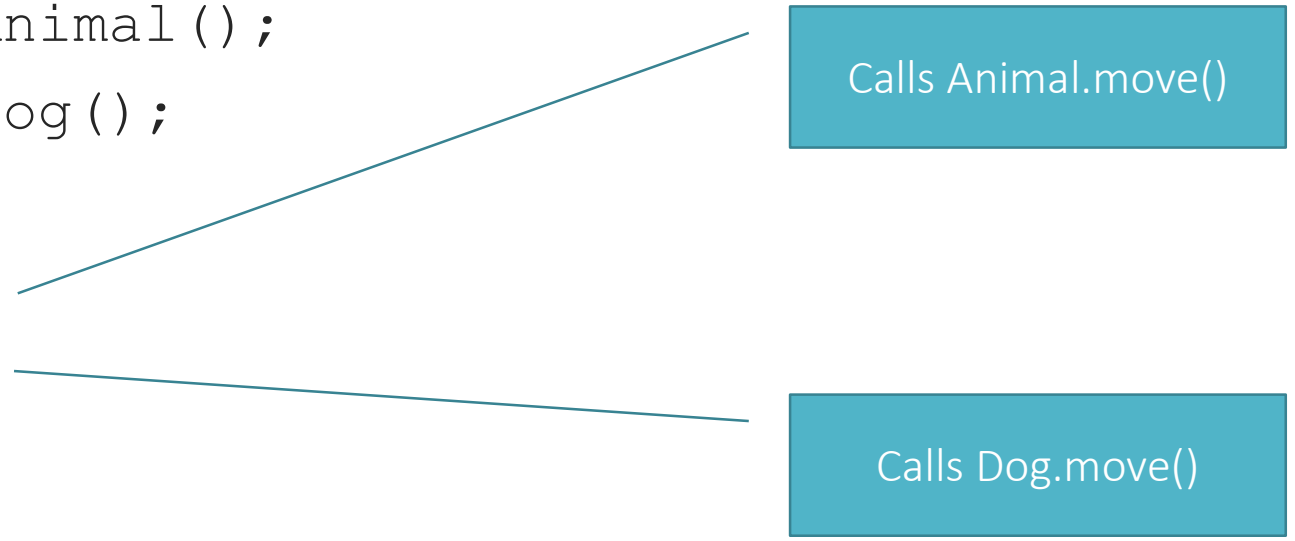
Even when the variable type is the superclass

Supported by **virtual method invocation**: method calls are dynamically dispatched based on the **runtime** type of the receiver object

Polymorphism example

```
public class TestAnimal {
      public static void main (String[] args) {
            Animal a1, a2;
            a1 = new Animal();
            a2 = new Dog();


            a1.move();
            a2.move();
      }
}
```



Constructors

A constructor is a special class member that is used to **create new objects** of that class

A class may have **many constructors** as long as they have different **signatures**

```
public Bicycle (int cadence, int speed, int gear) {
    this.gear = gear;
    this.cadence = cadence;
    this.speed = speed;
}

public Bicycle() {
    this.gear = 1;
    this.cadence = 0;
    this.speed = 0;
}
```

```
Bicycle bike1 = new Bicycle(30, 0, 0);
Bicycle bike2 = new Bicycle();
```

Constructors and inheritance

```
BankAccount(String name, int
initialAmount)
{
    this.name = name;
    this.balance = initialAmount;
    this.id = BankAccount.nextId++;
  }
```

Recall:

- Constructors look like a method with same name as class; no return type
- If no constructor is specified, a default **no-args** constructor is created

What about inheritance?

- "Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass." *(Java Tutorial)*

Also:

- "If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor of the superclass. If the super class does not have a no-argument constructor, you will get a compile-time error. `Object` does have such a constructor, so if `Object` is the only superclass, there is no problem."

Constructors and inheritance

A subclass **does not inherit** the superclass's constructors

… But can call them from its own constructor using `super`

```
public class MountainBike extends Bicycle {
    public int seatHeight;

    public MountainBike(int seatHeight, int cadence, int speed, int gear) {
        super(cadence, speed, gear);
        this.seatHeight = seatHeight;
    }
}
```

If you don't add a call to the superclass constructor in the subclass, a call to the "no-args" constructor is **automatically added.**

Constructor chaining

```
public class A {
  public A() { /*super();*/ System.out.println("A constructor"); }
}
public class B extends A {
  public B() { /*super();*/ System.out.println("B constructor"); }
}
public class C extends B {
  public C() { /*super();*/ System.out.println("C constructor"); }
}
```

The `this` keyword

Within an instance method or constructor, `this` is a reference to the **current object**

Why use `this.fieldName` instead of just `fieldName`?

- Most commonly: because a field is **shadowed** by a constructor or method parameter (especially in a setter)
- Other reasons:
  - *To pass the current object as a parameter to another method, e.g.,* `processInput(this);`
  - *To make clear that you are referring to a field*
  - *To call an alternative constructor (example coming up)*

```java
public class Foo {

    private String name;

    // ...

    public void setName(String name) {
        // This makes it clear that you are assigning
        // the value of the parameter "name" to the
        // instance variable "name".
        this.name = name;
    }

    // ...
}
```

http://stackoverflow.com/a/2411283

Calling alternative constructors

You can call another overloaded constructor from within the current constructor

Another possible use of the `this` keyword

```
public Bicycle (int cadence, int speed, int gear) {
    this.gear = gear;
    this.cadence = cadence;
    this.speed = speed;
}

public Bicycle() {
    this(0, 0, 1);
}
```

What about static methods?

Recall: Static methods are associated with a **class** (e.g., `Math.random()`)

If you provide a static method with the same signature in a subclass, it will **hide** (not override) the parent class method

- Why? Because in Java, polymorphism needs an **instance**, and static methods only have access to the **declared class**

What does this mean in practice?

- For **instance** (non-static) methods, Java will always execute the method in the subclass (polymorphism), whatever the run-time type
- For **static** methods, which method gets chosen depends on how it is called (i.e., which class is used)

Example (Java Tutorial)

```
public class Animal {

    public static void
testClassMethod() {
        System.out.println("The
static method in Animal");
    }
    public void
testInstanceMethod() {
        System.out.println("The
inst
    }
}
```

```
The static method in Animal
The instance method in Cat
```

```
public class Cat extends Animal {

    public static void testClassMethod() {
        System.out.println("The static method in Cat");
    }
    public void testInstanceMethod() {
        System.out.println("The instance method in Cat");
    }

    public static void main(String[] args) {
        Cat myCat = new Cat();
        Animal myAnimal = myCat;
        Animal.testClassMethod();
        myAnimal.testInstanceMethod();
    }
}
```

Rules for overriding/hiding

	Superclass instance method	Superclass static method
Subclass instance method	Overrides	Compile-time error
Subclass static method	Compile-time error	Hides

Access modifiers: the subclass method can allow **more**, but not **less**, access than the superclass method

	`public`	`protected`	*(default)*	`private`
`public`	•	•	•	
`protected`		•	•	*Not relevant – Private members cannot be inherited*
(default)			•	
`private`				

Accessing superclass methods

But what if you really want to use the non-overridden method sometimes?

Use the `super` keyword (similar to `this`, but refers to super-class implementation)

```
public class Superclass {

    public void printMethod() {

        System.out.println("Printed in
Superclass.");

    }

}
```

```
public class Subclass extends Superclass {

    // overrides printMethod in Superclass

    public void printMethod() {

        super.printMethod();

        System.out.println("Printed in
Subclass");

    }

    public static void main(String[] args) {

        Subclass s = new Subclass();

        s.printMethod();

    }

}
```

Method overloading

Overloading: a class can have multiple methods with the **same name**, as long as they have **different parameters**

```
public class DataArtist {

    ...

    public void draw(String s) {
        ...
    }

    public void draw(int i) {
        ...
    }

    public void draw(double f) {
        ...
    }

    public void draw(int i, double f) {
        ...
    }

}
```

Inheritance and static methods

Recall: Static methods are associated with a **class** (e.g., `Math.random()`)

If you provide a static method with the same signature in a subclass, it will **hide** (not override) the parent class method

- Why? Because in Java, polymorphism needs an **instance**, and static methods only have access to the **declared class**

What does this mean in practice?

- For **instance** (non-static) methods, Java will always execute the method in the subclass (polymorphism), whatever the run-time type
- For **static** methods, which method gets chosen depends on how it is called (i.e., which class is used)

Example (Java Tutorial)

```
public class Animal {

    public static void
testClassMethod() {
        System.out.println("The
static method in Animal");
    }
    public void
testInstanceMethod() {
        System.out.println("The
inst
    }
}
```

```
The static method in Animal
The instance method in Cat
```

```
public class Cat extends Animal {

    public static void testClassMethod() {
        System.out.println("The static method in Cat");
    }
    public void testInstanceMethod() {
        System.out.println("The instance method in Cat");
    }

    public static void main(String[] args) {
        Cat myCat = new Cat();
        Animal myAnimal = myCat;
        Animal.testClassMethod();
        myAnimal.testInstanceMethod();
    }
}
```

Rules for overriding/hiding

	Superclass instance method	Superclass static method
Subclass instance method	Overrides	Compile-time error
Subclass static method	Compile-time error	Hides

Access modifiers: the subclass method can allow **more**, but not **less**, access than the superclass method

	`public`	`protected`	*(default)*	`private`
`public`	•	•	•	
`protected`		•	•	*Not relevant – Private members cannot be inherited*
(default)			•	
`private`				

Accessing superclass methods

But what if you really want to use the non-overridden method sometimes?

Use the `super` keyword (similar to `this`, but refers to super-class implementation)

```
public class Superclass {

    public void printMethod() {

        System.out.println("Printed in
Superclass.");

    }

}
```

```
public class Subclass extends Superclass {

    // overrides printMethod in Superclass

    public void printMethod() {

        super.printMethod();

        System.out.println("Printed in
Subclass");

    }

    public static void main(String[] args) {

        Subclass s = new Subclass();

        s.printMethod();

    }

}
```

Method overloading

Overloading: a class can have multiple methods with the **same name**, as long as they have **different parameters**

```
public class DataArtist {

    ...
    public void draw(String s) {
        ...
    }
    public void draw(int i) {
        ...
    }
    public void draw(double f) {
        ...
    }
    public void draw(int i, double f) {
        ...
    }
}
```

Classes and inheritance up to now

A class gives a blueprint for objects of a particular type
  Fields (properties), methods (behaviours)

Subclasses **inherit from** the superclass to provide more specialised blueprints
  E.g., “Vehicle” -> “Bicycle” -> “MountainBike” -> “FullSuspensionMountainBike”

A subclass may **override** parent class methods to provide specialised behaviour

Issues:
  What if the behaviour only makes sense to implement in the subclasses?
    E.g., how would you define “move” at the level of Vehicle?
  What if you don’t want subclasses to override a method?
    E.g., in a security context or where behaviour must be predictable

Next time

Review of Objects, Classes, Inheritance

Abstract classes and methods

Final classes, methods, and fields

Exceptions

Using online resources