Java Programming 2 Inheritance: technical details

Mary Ellen Foster

MaryEllen.Foster@glasgow.ac.uk

Constructors revisited

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 **parameters**

```
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();
```

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);
}
```

Default (“no-args”) constructor

If you do not specify a constructor for your class, a default constructor is created

Properties of the default constructor

- Takes no parameters (“no-args”)
- Sets all fields to default values (0 for numeric types, false for Boolean, null for non-primitive types)
- Inserts a call to the parent class’s no-arg constructor – `super()`

This constructor is **only created if you do not specify your own constructor**

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

Constructors and inheritance

Situation so far:

- 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." *(Java Tutorial)*

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"); }
}
```

```
C c = new C();
```

```
A constructor
B constructor
C constructor
```

Constructors and inheritance

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

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

If you extend a class without a no-args constructor, you **have to** define a constructor for your subclass, and it **has to** call the correct parent constructor!

```
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 this line wasn't here, Java would automatically insert

`super();`

This would not compile!!!

Access modifiers and inheritance

Modifier	Same class	Same package	Any subclass	Any class
`public`	●	●	●	●
`protected`	●	●	●	
(default)	●	●		
`private`	●			

The subclass method can allow **more**, but not **less**, access than the superclass method

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

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 instance
method in Animal");
    }
}
```

```
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;
        myAnimal.testClassMethod();
        myAnimal.testInstanceMethod();
    }
}
```

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