Constructors

JUNE 30, 2015

Objects are constructed. You can never make a new object without invoking a constructor (and not just the constructor of the object's actual class type, but also the constructor of each of its superclasses).

Suppose there is a `Horse` class which extends `Animal` (and obviously `Animal` extends `Object`). Now this is how the constructors are called after `new Horse()` is invoked from `main()`:

1. main() calls new Horse()
2. Horse() calls super()
3. Animal() calls super()
4. Object()

Constructors **have no return type** and their **names must exactly match the class name**. Typically, constructors are used to initialize instance variable as below:

```
class Foo {
     int size;
     String name;

     Foo(String name, int size) {
        this.name = name;
        this.size = size;
     }
}
```

So when you invoke **`new Foo("rohit", 12)`**, an object of `Foo` class is created with `name` set as `Rohit` and `size` as `12`.

Below are the important rules for Constructors

1. If you don't type a constructor into your class code, a default constructor will be automatically generated by the compiler. The **default constructor (the one supplied by compiler) is ALWAYS a no-arg constructor**.

Class Code (WhatYouType)	Compiler Generated Constructor Code

`class Foo {`
 	
`}`
	`class Foo {`
	`  Foo() {`
	`  super();`
	`  }`
	`}`

2. Every constructor has, as its first statement, either a **call to an overloaded constructor (`this()`) or a call to the superclass constructor (`super()`)**, and if not it is inserted by the compiler. Remember, compiler **always inserts a *no-arg* call to `super()`**, it never passes any arguments.

`class Foo {`
`  Foo() { }`
`}`
	`class Foo {`
	`  Foo() {`
	`  super();`
	`  }`
	`}`

3. If you want a no-arg constructor and you've typed any other constructor(s) into your class code, the **compiler won't provide the no-arg constructor (or any other constructor)** for you. In other words, if you've typed in a constructor with arguments, you won't have a no-arg constructor unless you type it in yourself.

`class Foo {`
`  Foo(String s) {`
`  super();`
`  }`
`}`
	Code is the same. Compiler doesn't insert anything here.

4. Constructors can use **any access modifier, including private**. (A private constructor means only code within the class itself can instantiate an object of that type, so if the private constructor class wants to allow an instance of the class to be used, the class must provide a static method or variable that allows access to an instance created from within the class. Moreover, for a Singleton you are bound to make all constructors of class private.) Also note that the default constructor has the same access modifier as the class.

`public class Foo {`
`}`
	`public class Foo {`
	`  public Foo() {`
	`  super();`
	`  }`
	`}`

`private class Foo {`
`}`
	`private class Foo {`
	`  private Foo() {`
	`  super();`
	`  }`
	`}`

5. A call to `super()` can be either a *no-arg* call or can include arguments passed to the super constructor. But the compiler always inserts a *no-arg* call to `super()`. So there would be a problem in the below case:

```
class Animal {
    Animal(String name) { }
}
class Horse extends Animal {
    Horse() { }
}
```

```
class Animal {
    Animal(String name) { }
}
class Horse extends Animal {
    Horse() {
        super(); // problem!
    }
}
```

As there is no *no-arg* constructor in Animal class, the call to `super()` (inserted by the compiler) in `Horse` class will fail. In fact, the compiler won't even compile the code on the left hand side.

You can solve this in two ways, either you can provide a *no-arg* constructor in `Animal` class or you can yourself type `super("some name")` as first statement in the constructor in `Horse` class.

6. You cannot make a call to an instance method, or access an instance variable, until after the super constructor runs. **Only static variables and methods can be accessed as part of the call to super() or this()**.

```
1   public class Animal {
2       String name;
3
4       Animal(String name) {
5           this.name = name;
6       }
7
8       Animal() {
9           this(getName()); // ok!
10          this(getNickName()); // compiler error!
11          getNickName(); // ok!
12      }
13
14      static String getName() {
15          return "Horse";
16      }
17
18      String getNickName() {
19          return "Horsie";
20      }
21  }
```

Now let's understand the above code:

- line 9 is ok as the method `getName()` is static so it belongs to the class and can be accessed without the need of any object of the class.
- line 10 is not ok as the method `getNickName()` is not static and to access such a method you need an object first and from constructor chaining we know that until and unless super type constructors are called object isn't created.
- line 11 is fine because super type constructors are called before calling `getNickName()`.

Some more obvious rules which you already know but is worth mentioning

7. Abstract classes have constructors, and those constructors are always called when a concrete subclass is instantiated.

8. Interfaces do not have constructors. Interfaces are not part of an object's inheritance tree.

9. A **constructor can be invoked from within another constructor only**. You cannot invoke a constructor from anywhere else.

10. Lastly, **constructors are never inherited**. They aren't methods. **Constructors can't be overridden** (because they aren't methods and only instance methods can be overridden).

Q&A

Q1. What is the output of the below program?

```java
public class Animal {

    String name;

    Animal(String name) {
        this.name = name;
    }

    Animal() {
        this(getName());
    }

    String getName() {
        return "abc";
    }

    public static void main(String[] args) {
        Animal b = new Animal();
        System.out.println(b.name);
    }
}
```

A. abc

B. Empty string

C. Won’t compile

D. Runtime error

Q2. What is the result?

```
class Top {
    public Top(String s) {
        System.out.print("B");
    }
}

public class Bottom2 extends Top {
    public Bottom2(String s) {
        System.out.print("D");
    }

    public static void main(String[] args) {
        new Bottom2("C");
        System.out.println(" ");
    }
}
```

A. BD

B. DB

C. BDC

D. DBC

E. Compilation fails

Q3. What is the result?

```java
class Alpha {
    static String s = " ";

    protected Alpha() {
        s += "alpha ";
    }
}

class SubAlpha extends Alpha {
    private SubAlpha() {
        s += "sub ";
    }
}

public class SubSubAlpha extends Alpha {

    private SubSubAlpha() {
        s += "subsub ";
    }

    public static void main(String[] args) {
        new SubSubAlpha();
        System.out.println(s);
    }
}
```

A. subsub

B. sub subsub

C. alpha subsub

D. alpha sub subsub

E. Compilation fails

F. An exception is thrown at runtime

Q4.

```
class Building {
    Building() {
        System.out.print("b ");
    }

    Building(String name) {

        this();
        System.out.print("bn " + name);
    }
}

public class House extends Building {
    House() {
        System.out.print("h ");
    }

    House(String name) {
        this();
        System.out.print("hn " + name);

    }

    public static void main(String[] args) {
        new House("x ");
    }
}
```

What is the result?

A. h hn x
B. hn x h
C. b h hn x
D. b hn x h
E. bn x h hn x

F. b bn x h hn x
G. bn x b h hn x
H. Compilation fails

Overloading

« Previous

Statics

Next »