JAVAWORLD

NEWS

Understanding constructors

How constructors differ from methods

By Robert Nielsen

JavaWorld |
OCT 13, 2000 1:00 AM PT

Saying that a constructor is a method is like saying that the Australian platypus is just another mammal. To understand the platypus, it is important to know how it is different from other mammals. To understand the constructor, it is similarly important to understand how it differs from a method. Any student of Java, especially one studying for certification, needs to know those differences; in this article, I will concretely spell them out. Table 1, at the end of this article, summarizes the key constructor/method distinctions.

Purpose and function

Constructors have one purpose in life: to create an instance of a class. This can also be called creating an object, as in:

```
Platypus p1 = new Platypus();
```

The purpose of methods, by contrast, is much more general. A method's basic function is to execute Java code.

Signature differences

Constructors and methods differ in three aspects of the signature: modifiers, return type, and name. Like methods, constructors can have any of the access modifiers: public, protected, private, or none (often called *package* or *friendly*). Unlike methods, constructors can take only access modifiers. Therefore, constructors cannot be `abstract`, `final`, `native`, `static`, or `synchronized`.

The return types are very different too. Methods can have any valid return type, or no return type, in which case the return type is given as `void`. Constructors have no return type, not even `void`.

Finally, in terms of the signature, methods and constructors have different names. Constructors have the same name as their class; by convention, methods use names other than the class name. If the Java program follows normal conventions, methods will start with a lowercase letter, constructors with an uppercase letter. Also, constructor names are usually nouns because class names are usually nouns; method names usually indicate actions.

The use of "this"

Constructors and methods use the keyword `this` quite differently. A method uses `this` to refer to the instance of the class that is executing the method. Static methods do not use `this`; they do not belong to a class instance, so `this` would have nothing to reference. Static methods belong to the class as a whole, rather than to an instance. Constructors use `this` to refer to another constructor in the same class with a different parameter list. Study the following code:

```
public class Platypus {
	String name;
	Platypus(String input) {
		name = input;
	}
	Platypus() {
		this("John/Mary Doe");
	}
	public static void main(String args[]) {
		Platypus p1 = new Platypus("digger");
		Platypus p2 = new Platypus();
	}
}
```

In the code, there are two constructors. The first takes a `String` input to name the instance. The second, taking no parameters, calls the first constructor by the default name `"John/Mary Doe"`.

If a constructor uses `this`, it must be in the constructor's first line; ignoring this rule will cause the compiler to object.

The use of "super"

Methods and constructors both use `super` to refer to a superclass, but in different ways. Methods use `super` to execute an overridden method in the superclass, as the following example illustrates:

```
class Mammal {
       void getBirthInfo() {
              System.out.println("born alive.");
       }
}
class Platypus extends Mammal {
       void getBirthInfo() {
              System.out.println("hatch from eggs");
              System.out.print("a mammal normally is ");
              super.getBirthInfo();
       }
}
```

In the above program, the call to `super.getBirthInfo()` calls the overridden method of the `Mammal` superclass.

Constructors use `super` to invoke the superclass's constructor. If a constructor uses `super`, it must use it in the first line; otherwise, the compiler will complain. An example follows:

```
public class SuperClassDemo {
        SuperClassDemo() {}
}
class Child extends SuperClassDemo {
       Child() {
              super();
       }
}
```

In the above (and trivial!) example, the constructor `Child()` includes a call to `super`, which causes the class `SuperClassDemo` to be instantiated, in addition to the `Child` class.

Compiler-supplied code

The new Java programmer may stumble when the compiler automatically supplies code for constructors. This happens if you write a class with no constructors; the compiler will automatically supply a no-argument constructor for you. Thus, if you write:

```
public class Example {}
```

it is functionally equivalent to writing:

```
public class Example {
	Example() {}
}
```

The compiler also automatically supplies code when you do not use `super` (using zero or more parameters) as the first line of a constructor. In this case, the computer automatically inserts `super`. Thus, if you write:

```
public class TestConstructors {
	TestConstructors() {}
}
```

it is functionally equivalent to writing:

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

The sharp-eyed beginner may wonder how the above program can call the parent class's constructor when `TestConstructor` is not extending any class. The answer is that Java extends the `Object` class when you do not explicitly extend a class. The compiler automatically supplies a no-argument constructor if no constructor is explicitly declared, and automatically supplies a no-argument `super` call when a constructor has no explicit call to `super`. So the following two code snippets are

functionally equivalent:

```
public class Example {}
```

and

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

Inheritance

What is wrong with the following scenario? A lawyer is reading the will of `A. Class`. Members of the `Class` family are gathered around a large conference table, some sobbing gently. The lawyer reads, "I, `A. Class`, being of sound mind and body, leave all my constructors to my children."

The problem is that constructors cannot be inherited. Fortunately for the `Class` children, they will automatically inherit any of their parents' methods, so the `Class` children will not become totally destitute.

Remember, Java methods are inherited, constructors are not. Consider the following class:

```
public class Example {
        public void sayHi {
                system.out.println("Hi");
        }
        Example() {}
}
public class SubClass extends Example {
}
```

The `SubClass` class automatically inherits the `sayHi` method found in the parent class. However, the constructor `Example()` is not inherited by the `SubClass`.

Summarizing the differences

Just as the platypus differs from the typical mammal, so too do constructors differ from methods; specifically in their purpose, signature, and use of `this` and `super`. Additionally, constructors differ with respect to inheritance and compiler-supplied code. Keeping all these details straight can be a chore; the following table provides a convenient summary of the salient points. You can find more information regarding constructors and methods in the Resources section below.

Table 1. Differences Between Constructors and Methods

Topic	Constructors	Methods
Purpose	Create an instance of a class	Group Java statements
Modifiers	Cannot be `abstract`, `final`, `native`, `static`, or `synchronized`	Can be `abstract`, `final`, `native`, `static`, or `synchronized`
Return type	No return type, not even `void`	`void` or a valid return type
Name	Same name as the class (first letter is capitalized by convention) -- usually a noun	Any name except the class. Method names begin with a lowercase letter by convention -- usually the name of an action
`this`	Refers to another constructor in the same class. If used, it must be the first line of the constructor	Refers to an instance of the owning class. Cannot be used by static methods
`super`	Calls the constructor of the parent class. If used, must be the first line of the constructor	Calls an overridden method in the parent class

Inheritance	Constructors are not inherited	Methods are inherited
Compiler automatically supplies a default constructor	If the class has no constructor, a no-argument constructor is automatically supplied	Does not apply
Compiler automatically supplies a default call to the superclass constructor	If the constructor makes no zero-or-more argument calls to `super`, a no-argument call to `super` is made	Does not apply

Robert Nielsen is a Sun Certified Java 2 Programmer. He holds a master's degree in education, specializing in computer-assisted instruction, and has taught in the computer field for several years. He has also published computer-related articles in a variety of magazines.

Learn more about this topic

- Some books that cover the basics of constructors and methods are
- *The Complete Java 2 Study Certification Guide*, Simon Roberts et al. (Sybex, 2000) http://www.amazon.com/exec/obidos/ASIN/0782128254/qid=969399182/sr=1-2/102-9220485-9634548 (http://www.amazon.com/exec/obidos/ASIN/0782128254/qid=969399182/sr=1-2/102-9220485-9634548)
- *Java 2 (Exam Cram),* Bill Brogden (The Coriolis Group, 1999): http://www.amazon.com/exec/obidos/ASIN/1576102912/qid%3D969399279/102-9220485-9634548 (http://www.amazon.com/exec/obidos/ASIN/1576102912/qid%3D969399279/102-9220485-9634548)
- *Java in a Nutshell,* Davis Flanagan (O'Reilly & Associates, 1999)http://www.amazon.com/exec/obidos/ASIN/1565924878/o/qid=969399378/sr=2-1/102-9220485-9634548 (http://www.amazon.com/exec/obidos/ASIN/1565924878/o/qid=969399378/sr=2-1/102-9220485-9634548)
- Visit the Sun Microsystems Website for more coverage of methods and constructors http://java.sun.com/docs/books/tutorial/trailmap.html (http://java.sun.com/docs/books/tutorial/trailmap.html)
- For more Java content for beginners, read *JavaWorld*'s new **Java 101** column series http://www.javaworld.com/javaworld/topicalindex/jw-ti-java101.html (http://www.javaworld.com/javaworld/topicalindex/jw-ti-java101.html)