

Now that we've covered fields and methods, we move on to other important members of a class. Constructors and initializers are class members whose job is to initialize the fields of a class.

Take another look at how we've been creating `Circle` objects:

```
Circle c = new Circle();
```

What are those parentheses doing there? They make it look like we're calling a method. In fact, that is exactly what we're doing. Every class in Java has at least one *constructor*, which is a method that has the same name as the class and whose purpose is to perform any necessary initialization for a new object. Since we didn't explicitly define a constructor for our `Circle` class in Example 3-1, Java gave us a default constructor that takes no arguments and performs no special initialization.

Here's how a constructor works. The `new` operator creates a new, but uninitialized, instance of the class. The constructor method is then called, with the new object passed implicitly (a `this` reference, as we saw earlier) as well as whatever arguments that are specified between parentheses passed explicitly. The constructor can use these arguments to do whatever initialization is necessary. There is some obvious initialization we could do for our circle objects, so let's define a constructor. Example 3-2 shows a new definition for `Circle` that contains a constructor that lets us specify the radius of a new `Circle` object. The constructor also uses the `this` reference to distinguish between a method parameter and an instance field of the same name.

Example 3-2. A constructor for the `Circle` class

```
public class Circle {
    public static final double PI = 3.14159; // A constant
    public double r; // An instance field that holds the radius of the circle

    // The constructor method: initialize the radius field
    public Circle(double r) { this.r = r; }

    // The instance methods: compute values based on the radius
    public double circumference() { return 2 * PI * r; }
    public double area() { return PI * r*r; }
}
```

When we relied on the default constructor supplied by the compiler, we had to write code like this to initialize the radius explicitly:

```
Circle c = new Circle();
c.r = 0.25;
```

With this new constructor, the initialization becomes part of the object creation step:

```
Circle c = new Circle(0.25);
```

- The constructor name is always the same as the class name.
- Unlike all other methods, a constructor is declared without a return type, not even `void`.
- The body of a constructor should initialize the `this` object.
- A constructor may not return `this` or any other value. A constructor may include a return statement, but only one that does not include a return value.

Sometimes you want to initialize an object in a number of different ways, depending on what is most convenient in a particular circumstance. For example, we might want to initialize the radius of a circle to a specified value or a reasonable default value. Since our `Circle` class has only a single instance field, we can't initialize it too many ways, of course. But in more complex classes, it is often convenient to define a variety of constructors. Here's how we can define two constructors for `Circle`:

```
public Circle() { r = 1.0; }
public Circle(double r) { this.r = r; }
```

A specialized use of the `this` keyword arises when a class has multiple constructors; it can be used from a constructor to invoke one of the other constructors of the same class. In other words, we can rewrite the two previous `Circle` constructors as follows:

```
// This is the basic constructor: initialize the radius
public Circle(double r) { this.r = r; }
// This constructor uses this() to invoke the constructor above
public Circle() { this(1.0); }
```

The `this()` syntax is a method invocation that calls one of the other constructors of the class. The particular constructor that is invoked is determined by the number and type of arguments, of course. This is a useful technique when a number of constructors share a significant amount of initialization code, as it avoids repetition of that code. This would be a more impressive example, of course, if the one-parameter version of the `Circle()` constructor did more initialization than it does.

The `Circle` defined earlier is a simple class that distinguishes circle objects only by their radii. Suppose, instead, that we want to represent circles that have both a size and a position. For example, a circle of radius 1.0 centered at point 0,0 in the Cartesian plane is different from the circle of radius 1.0 centered at point 1,2. To do this, we need a new class, which we'll call `PlaneCircle`. We'd like to add the ability to represent the position of a circle without losing any of the existing functionality of the `Circle` class. This is done by defining `PlaneCircle` as a subclass of `Circle` so that `PlaneCircle` inherits the fields and methods of its superclass, `Circle`. The ability to add functionality to a class by subclassing, or extending, is central to the object-oriented programming paradigm.

```
public class PlaneCircle extends Circle {
    // We automatically inherit the fields and methods of Circle,
    // so we only have to put the new stuff here.
    // New instance fields that store the center point of the circle
    public double cx, cy;

    // A new constructor method to initialize the new fields
    // It uses a special syntax to invoke the Circle() constructor
    public PlaneCircle(double r, double x, double y) {
        super(r);    // Invoke the constructor of the superclass, Circle()
        this.cx = x;  // Initialize the instance field cx
        this.cy = y;  // Initialize the instance field cy
    }

    // The area() and circumference() methods are inherited from Circle
    // A new instance method that checks whether a point is inside the circle
    // Note that it uses the inherited instance field r
    public boolean isInside(double x, double y) {
        double dx = x - cx, dy = y - cy;    // Distance from center
        double distance = Math.sqrt(dx*dx + dy*dy); // Pythagorean theorem
        return (distance < r);              // Returns true or false
    }
}
```

Note the use of the keyword `extends` in the first line of Example 3-3. This keyword tells Java that `PlaneCircle` extends, or subclasses, `Circle`, meaning that it inherits the fields and methods of that class.* The definition of the `isInside()` method shows field inheritance; this method uses the field `r` (defined by the `Circle` class) as if it were defined right in `PlaneCircle` itself. `PlaneCircle` also inherits the methods of `Circle`. Thus, if we have a `PlaneCircle` object referenced by variable `pc`, we can say:

```
double ratio = pc.circumference() / pc.area();
```

This works just as if the `area()` and `circumference()` methods were defined in `PlaneCircle` itself.

This constructor explicitly initializes the `cx` and `cy` fields newly defined by `PlaneCircle`, but it relies on the superclass `Circle()` constructor to initialize the inherited fields of the class. To invoke the superclass constructor, our constructor calls `super()`. `super` is a reserved word in Java. One of its uses is to invoke the constructor method of a superclass from within the constructor method of a subclass. This use is analogous to the use of `this()` to invoke one constructor method of a class from within another constructor method of the same class. Invoking a constructor using `super()` is subject to the same restrictions as is using `this()`:

- `super()` can be used in this way only within a constructor method.
- The call to the superclass constructor must appear as the first statement within the constructor method, even before local variable declarations.

Java guarantees that the constructor method of a class is called whenever an instance of that class is created. It also guarantees that the constructor is called whenever an instance of any subclass is created. In order to guarantee this second point, Java must ensure that every constructor method calls its superclass constructor method. Thus, if the first statement in a constructor does not explicitly invoke another constructor with `this()` or `super()`, Java implicitly inserts the call `super()`, that is, it calls the superclass constructor with no arguments. If the superclass does not have a constructor that takes no arguments, this implicit invocation causes a compilation error.

Consider what happens when we create a new instance of the `PlaneCircle` class. First, the `PlaneCircle` constructor is invoked. This constructor explicitly calls `super(r)` to invoke a `Circle` constructor, and that `Circle()` constructor implicitly calls `super()` to invoke the constructor of its superclass, `Object`. The body of the `Object` constructor runs first. When it returns, the body of the `Circle()` constructor runs. Finally, when the call to `super(r)` returns, the remaining statements of the `PlaneCircle()` constructor are executed.

What all this means is that constructor calls are chained; any time an object is created, a sequence of constructor methods is invoked, from subclass to super-

class on up to `Object` at the root of the class hierarchy. Because a superclass constructor is always invoked as the first statement of its subclass constructor, the body of the `Object` constructor always runs first, followed by the constructor of its subclass and on down the class hierarchy to the class that is being instantiated. There is an important implication here; when a constructor is invoked, it can count on the fields of its superclass to be initialized.