

Declarations and Definitions

Sequences of tokens are used to construct the next building blocks of Java classes: declarations and definitions. A class declaration provides the name and visibility of a class. For our example,

```
public class Distance
```

is the class declaration. It consists (in this case) of two keywords, `public` and `class` followed by the identifier `Distance`.

This means that we are defining a class named `Distance`. Other classes, or in our case, the command line, can refer to the class by this name. The `public` keyword is an access modifier which declares that this class and its members may be accessed from other classes. The `class` keyword, obviously, identifies this declaration as a class. Java also allows declarations of *interfaces* and (as of Java 5) *annotations*.

The class declaration is then followed by a block (surrounded by curly braces) which provides the class's definition. The definition is the implementation of the class - the declaration and definitions of the class's members. This class contains exactly six members, which we will explain in turn.

1. Two field declarations, named `point0` and `point1`
2. A constructor declaration
3. Three method declarations

Example: Instance Fields

The declaration

```
private java.awt.Point point0, point1;
```

declares two *instance fields*. Instance fields represent named values that are allocated whenever an instance of the class is constructed. When a Java program creates a `Distance` instance, that instance will contain space for `point0` and `point1`. When another `Distance` object is created, it will contain space for its *own* `point0` and `point1` values. The value of `point0` in the first `Distance` object can vary independently of the value of `point0` in the second `Distance` object.

This declaration consists of:

1. The `private` access modifier, which means these instance fields are not visible to other classes.
2. The type of the instance fields. In this case, the type is `java.awt.Point`. This is the class `Point` in the `java.awt` package. The names of the instance fields in a comma separated list.

These two fields could also have been declared with two separate but more verbose declarations,

```
private java.awt.Point point0;  
private java.awt.Point point1;
```

Since the types of these fields is a reference type (i.e. a field that *refers to* or can hold a *reference to* an object value), Java will implicitly initialize the values of `point0` and `point1` to `null` when a `Distance` instance is created. The `null` value means that a reference value does not refer to an object. The special Java literal, `null` is used to represent the null value in a program. While you can explicitly assign `null` values in a declaration, as in

```
private java.awt.Point point0 = null;  
private java.awt.Point point1 = null;
```

it is not necessary and most programmers omit such default assignments.

Example: Constructor

A *constructor* is a special method in a class which is used to construct an instance of the class. The constructor can perform initialization for the object, beyond that which the Java VM does automatically. For example, Java will automatically initialize the fields `point0` and `point1` to null.

Below is the constructor for this class. It consists of five parts:

1. The optional access modifier(s). In this case, the constructor is declared `public`
2. The constructor name, which must match the class name exactly: `Distance` in this case.
3. The constructor parameters. The parameter list is required. Even if a constructor does not have any parameters, you must specify the empty list `()`. The parameter list declares the type and name of each of the method's parameters.
4. An optional **throws** clause which declares the exceptions that the constructor may throw. This constructor does not declare any exceptions.
5. The constructor body, which is a Java block (enclosed in `{ }`). This constructor's body contains two statements.

`public Distance(int x0, int y0, int x1, int y1) { point0 = new java.awt.Point(x0, y0); point1 = new java.awt.Point(x1, y1); }` This constructor accepts four parameters, named `x0`, `y0`, `x1` and `y1`. Each parameter requires a parameter type declaration, which in this example is **int** for all four parameters. Java integer values are signed, 32 bit two's complement integers. The parameters in the parameter list are separated by commas.

The two assignments in this constructor use Java's *new operator* to allocate two `java.awt.Point` objects. The first allocates an object representing the first point, `(x0, y0)`, and assigns it to the `point0` instance variable (replacing the null value that the instance variable was initialized to). The second statement allocates a second `java.awt.Point` instance with `(x1, y1)` and assigns it to the `point1` instance variable.

This is the constructor for the `Distance` class. `Distance` implicitly extends from `java.lang.Object`. Java inserts a call to the super constructor as the first executable statement of the constructor if there is not one explicitly coded. The above constructor body is equivalent to the following body with the explicit super constructor call:

```
{
    super();
    point0 = new java.awt.Point(x0, y0);
    point1 = new java.awt.Point(x1, y1);
}
```

While it is true that this class could be implemented in other ways, such as simply storing the coordinates of the two points and computing the distance as $\sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2}$, this class instead uses the existing `java.awt.Point` class. This choice matches the abstract definition of this class: to print the distance between two points on the plane. We take advantage of existing behavior already implemented in the Java platform rather than implementing it again. We will see later how to make the program more flexible without adding much complexity, because we choose to use object abstractions here. However, the key point is that this class uses information hiding. That is, *how* the class stores its state or how it computes the distance is hidden. We can change this implementation without altering how clients use and invoke the class.

Example: Methods

Methods are the third and most important type of class member. This class contains three *methods* in which the behavior of the Distance class is defined: `printDistance()`, `main()`, and `intValue()`

The `printDistance()` method

The `printDistance()` method prints the distance between the two points to the standard output (normally the console).

```
public void printDistance()
{
    System.out.println("Distance between " + point0
                      + " and " + point1
                      + " is " + point0.distance(point1));
}
```

This *instance method* executes within the context of an implicit Distance object. The instance field references, `point0` and `point1`, refer to instance fields of that implicit object. You can also use the special variable `this` to explicitly reference the current object. Within an instance method, Java binds the name `this` to the object on which the method is executing, and the type of `this` is that of the current class. The body of the `printDistance` method could also be coded as `System.out.println("Distance between " + this.point0 + " and " + this.point1 + " is " + this.point0.distance(this.point1));`

to make the instance field references more explicit.

This method both computes the distance and prints it in one statement. The distance is computed with `point0.distance(point1)`; `distance()` is an instance method of the `java.awt.Point` class (of which `point0` and `point1` are instances. The method operates on `point0` (binding `this` to the object that `point0` refers to during the execution of the method) and accepting another `Point` as a parameter. (Actually, it is slightly more complicated than that, but we'll explain later.) The result of the `distance()` method is a double precision floating point number.

This method uses the syntax

```
"Distance between " + this.point0
+ " and " + this.point1
+ " is " + this.point0.distance(this.point1)
```

to construct a `String` to pass to the `System.out.println()`. This expression is a series of *String concatenation* methods which concatenates `Strings` or the `String` representation of primitive types (such as doubles) or objects, and returns a long string. For example, the result of this expression for the points (0,3) and (4,0) is the `String`

```
"Distance between java.awt.Point[x=0,y=3] and java.awt.Point[x=4,y=0]
is 5.0"
```

which the method then prints to `System.out`.

In order to print, we invoke the `println()`. This is an instance method from `java.io.PrintStream`, which is the type of the static field `out` in the class `java.lang.System`. The Java VM binds `System.out` to the standard output stream when it starts a program.