Advertisement: Support JavaWorld, click here!

SunWorld SUBSCRIBE HERE! IDG's Web-only magazine for the Unix professional

July 2000

JavaWorld

FUELING INNOVATION

Java 101

Learn how to store data in objects

The journey from Java wanna-be to Java developer continues

Summary
In this second installment of **Java 101**, Jacob Weintraub delves into storing data in Java and the various ways you can use that data. Specifically, he examines how objects store data and how you can pass data to objects in method calls. He also presents a discussion on encapsulation, a basic object-oriented design strategy that helps you write better code. *(1,900 words)*

By Jacob Weintraub

In the previous **Java 101** column, I discussed Java as an interpreted language and explained Java byte code and the Java Virtual Machine. You learned how to set up Java on a system using the Java 2 SDK, downloadable from Sun, and created your first simple Java program. I then covered the basics of object-oriented programming, and how it is centered around types and objects. Finally, I introduced the class construct in Java, and you created a class that looks like this:

```
public class AlarmClock {
  public void snooze() {{column,
    System.out.println("ZZZZZ");
  }
}
```

You also created a program (a class with a main method) that tests the `AlarmClock` class:

```
public class AlarmClockTest {
  public static void main(String[] args) {
    AlarmClock aClock = new AlarmClock();
    aClock.snooze();
  }
}
```

If you didn't read the last column, you'll probably want to review it before tackling this one.

Variables and primitive types

Though the snooze button is probably the most commonly used button on an alarm clock, the simple `AlarmClock` class you've created so far is still missing some important requirements. For instance, you have no way of manipulating how long the alarm clock will stay in snooze mode. However, before you can do that, you must take a more detailed look at how Java controls data.

Developers use variables in Java to hold data, with all variables having a data type and a name. The data type determines the values that a variable can hold. You will see in the examples below how integral types hold whole numbers, floating point types hold real numbers, and string types hold character strings.

Called primitive types, integral and floating point are the simplest data types that Java uses. The following program illustrates the integral type, which can hold both positive and negative whole numbers. This program also illustrates comments, which document your code but don't affect the program in any way.

```
/*
  * This is also a comment. The compiler ignores everything from
  * the first /* until a "star slash" which ends the comment.
  *
  * Here's the "star slash" that ends the comment.
  */

public class IntegerTest {
  public static void main(String[] args) {

    // Here's the declaration of an int variable called anInteger,
    // which you give an initial value of 100.

    int anInteger = 100; // Declare and initialize anInteger
    System.out.println(anInteger); // Outputs 100

    // You can also do arithmetic with primitive types, using the
    // standard arithmetic operators.

    anInteger = 100 + 100;
    System.out.println(anInteger); // Outputs 200
  }
}
```

Java also uses floating point types, which can hold real numbers (numbers that include a decimal place).

Here is an example program:

```
public class DoubleTest {
  public static void main(String[] args) {

    // Here's the declaration of a double variable called aDouble.
    // You also give aDouble an initial value of 5.76.

    double aDouble = 5.76; // Declare and initialize aDouble
    System.out.println(aDouble); // Outputs 5.76

    // You can also do arithmetic with floating point types.

    aDouble = 5.76 + 1.45;
    System.out.println(aDouble); // Outputs 7.21
  }
}
```

Try running the programs above. Remember, you have to compile them before you can run them:

```
javac *.java
java IntegerTest
java DoubleTest
```

Java uses four integral types and two floating point types, which both hold different ranges of numbers and take up varying amounts of storage space. The following table lists them, along with some of their properties:

Integral types		
Type	**Size (Bits)**	**Range**
byte	8	-128 to 127
short	16	-32,768 to 32,767
int	32	-2,147,483,648 to 2,147,483,647
long	64	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

Floating point types		
Type	**Size (Bits)**	**Range**
float	32	Single precision floating point (IEEE 754 conforming)
double	64	Double precision floating point (IEEE 754 conforming)

Integral and floating point types

A string type holds strings, and handles them differently from the way integral and floating point types handle numbers. The Java language includes a `String` class to represent strings. You declare a string using the type `String`, and initialize it with a quoted string, a sequence of characters contained within double quotes, as shown below. You can also combine two strings using the `+` operator.

```
// Code fragment
// Declaration of variable s of type String,
// and initialization with quoted string "Hello."
String s = "Hello";
// Concatenation of string in s with quoted string " World"
String t = s + " World";
System.out.println(t); // Outputs Hello World
```

Variable scope

In addition to type, *scope* is also an important characteristic of a variable. Scope establishes when a variable is created and destroyed and where a developer can access the variable within a program. The place in your program where you declare the variable determines its scope.

So far, I've discussed *local variables*, which hold temporary data that you use within a method. You declare local variables inside methods, and you can access them only from within those methods. This means that you can retrieve only local variables `anInteger`, which you used in `IntegerTest`, and `aDouble`, which you used in `DoubleTest`, from the main method in which they were declared and nowhere else.

You can declare local variables within any method. The example code below declares a local variable in the `AlarmClock snooze()` method:

```
public class AlarmClock {
     public void snooze() {
         // Snooze time in millisecond = 5 secs
         long snoozeInterval = 5,000;
         System.out.println("ZZZZZ for: " + snoozeInterval);
     }
}
```

You can get to `snoozeInterval` only from the `snooze()` method, which is where you declared `snoozeInterval`. Refer to the example below:

```
public class AlarmClockTest {
  public static void main(String[] args) {
    AlarmClock aClock = new AlarmClock();
    aClock.snooze(); // This is still fine.
    // The next line of code is an ERROR.
    // You can't access snoozeInterval outside the snooze method.
    snoozeInterval = 10,000;
  }
}
```

Method parameters

A method parameter, which has a scope similar to a local variable, is another type of variable. Method parameters pass arguments into methods. When you declare the method, you specify its arguments in a parameter list. You pass the arguments when you call the method. Method parameters function similarly to local variables in that they lie within the scope of the method to which they are linked, and can be used throughout the method. However, unlike local variables, method parameters obtain a value from the caller when it calls a method. Here's a modification of the alarm clock that allows you to pass in the `snoozeInterval`.

```
public class AlarmClock {
  public void snooze(long snoozeInterval) {
    System.out.println("ZZZZZ for: " + snoozeInterval);
  }
}

public class AlarmClockTest {
  public static void main(String[] args) {
    AlarmClock aClock = new AlarmClock();
    // Pass in the snooze interval when you call the method.
    aClock.snooze(10,000); // Snooze for 10,000 msecs.
  }
}
```

Member variables -- how objects store data

Local variables are useful, but because they provide only temporary storage, their value is limited. Since their lifetimes span the length of the method in which they are declared, local variables compare to a notepad that appears every time you receive a telephone call, but disappears once you hang up the phone. That setup can be useful for jotting down notes, but you often want something a little more permanent. What's a programmer to do? Enter *member variables*.

Member variables -- of which there are two, *instance* and *static* -- make up part of a class. I will consider instance variables now, and return to static variables in a later article.

Developers implement instance variables to contain data useful to a class. An instance variable differs from a local variable in the nature of its scope and its lifetime. The entire class makes up the scope of an instance variable, not the method in which it was declared. In other words, developers can access instance variables anywhere in the class. In addition, the lifetime of an instance variable does not depend on any particular method of the class; that is, its lifetime is the lifetime of the instance that contains it.

Remember instances from the previous article? Instances are the actual objects that you create from the blueprint you design in the class definition. You declare instance variables in the class definition, affecting each instance you create from the blueprint. Each instance contains those instance variables, and data held within the variables can vary from instance to instance.

Consider the `AlarmClock` class. Passing the `snoozeInterval` into the `snooze()` method isn't a great design. Imagine having to type in a snooze interval on your alarm clock each time you fumbled for the snooze button. Instead, just give the whole alarm clock a `snoozeInterval`. You complete this with an instance variable in the `AlarmClock` class, as shown below:

```
public class AlarmClock {

  // You declare snoozeInterval here. This makes it an instance variable.
  // You also initialize it here.
  long m_snoozeInterval = 5,000; // Snooze time in millisecond = 5 secs.


  public void snooze() {
    // You can still get to m_snoozeInterval in an AlarmClock method
    // because you are within the scope of the class.
    System.out.println("ZZZZZ for: " + m_snoozeInterval);
   }
}
```

You can access instance variables almost anywhere within the class that declares them. To be technical about it, you declare the instance variable within the *class scope*, and you can retrieve it from almost anywhere within that scope. Practically speaking, you can access the variable anywhere between the first curly bracket that starts the class and the closing bracket. Since you also declare methods within the class scope, they too can access the instance variables.

You can also access instance variables from outside the class, as long as an instance exists, and you have a variable that references the instance. To retrieve an instance variable through an instance, you use the *dot operator* together with the instance. That may not be the ideal way to access the variable, but for now, complete it this way for illustrative purposes:

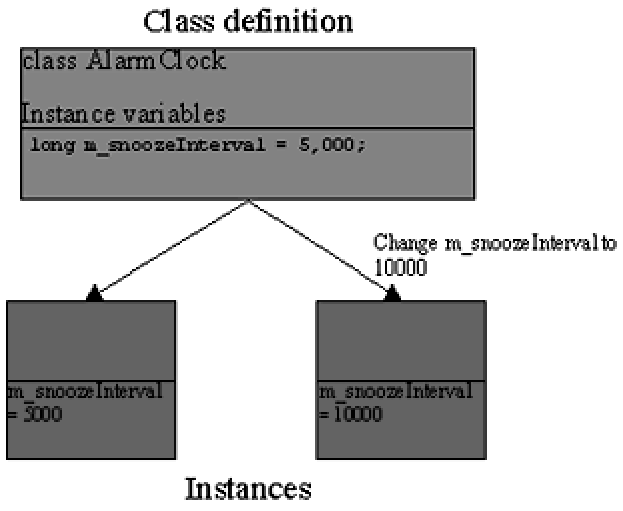
```
public class AlarmClockTest {
  public static void main(String[] args) {
    // Create two clocks. Each has its own m_snoozeInterval
    AlarmClock aClock1 = new AlarmClock();
    AlarmClock aClock2 = new AlarmClock();

    // Change aClock2
    // You'll soon see that there are much better ways to do this.
    aClock2.m_snoozeInterval = 10,000;

    aClock1.snooze(); // Snooze with aClock1's interval
    aClock2.snooze(); // Snooze with aClock2's interval
  }
}
```

Try this program out, and you'll see that `aClock1` still has its interval of 5,000 while `aClock2` has an interval of 10,000. Again, each instance has its own instance data.

Don't forget, the class definition is only a blueprint, so the instance variables don't actually exist until you create instances from the blueprint. Each instance of a class has its own copy of the instance variables, and the blueprint defines what those instance variables will be.



Two instances of AlarmClock with their own instance variables

Encapsulation

Encapsulation remains as one of the foundations of object-oriented programming. When using encapsulation, the user interacts with the type through the exposed behavior, not directly with the internal implementation. Through encapsulation, you hide the details of a type's implementation. In Java, encapsulation basically translates to this simple guideline: "Don't access your object's data directly; use its methods."

That is an elementary idea, but it eases our lives as programmers. Imagine, for example, that you wanted to instruct a "person" object to stand up. Without encapsulation, your commands could go something like this: "Well, I guess you'd need to tighten this muscle here at the front of the leg, loosen this muscle here at the back of the leg. Hmmm -- need to bend at the waist too. Which muscles spark that movement? Need to tighten these, loosen those. Whoops! Forgot the other leg. Darn. Watch it -- don't tip over ..." You get the idea. With encapsulation, you would just need to invoke the `standUp()` method. Pretty easy, yes?

Some advantages to encapsulation:

- **Abstraction of detail**: The user interacts with a type at a higher level. If you use the `standUp()` method, you no longer need to know all the muscles required to initiate that motion.

- **Isolation from changes**: Changes in internal implementation don't affect the users. If a person sprains an ankle, and depends on a cane for a while, the users still invoke only the `standUp()` method.

- **Correctness**: Users can't arbitrarily change the insides of an object. They can only complete what you allow them to do in the methods you write.

Here is a short example in which encapsulation clearly helps in a program's accuracy:

```
// Bad -- doesn't use encapsulation
public class Person {
  int m_age;
}

public class PersonTest {
  public static void main(String[] args) {
    Person p = new Person();
    p.m_age = -5; // Hey -- how can someone be minus 5 years old?
  }
}

// Better - uses encapsulation
public class Person {
  int m_age;

  public void setAge(int age) {
    // Check to make sure age is greater than 0. I'll talk more about
    // if statements at another time.
    if (age > 0) {
      m_age = age;
    }
  }
}

public class PersonTest {
  public static void main(String[] args) {
    Person p = new Person();
    p.setAge(-5); // Won't have any effect now.
  }
}
```

Even that simple program shows how you can slip into trouble if you directly access the internal data of classes. The larger and more complex the program, the more important encapsulation becomes. And remember, many programs start out small and then grow to last indefinitely, so design them correctly, right from the beginning. To apply encapsulation to `AlarmClock`, you can just create methods to manipulate the snooze interval.

Before I proceed, I should discuss methods in more detail. Methods can return values that the caller uses. To return a value, declare a nonvoid return type, and use a `return` statement. The `getSnoozeInterval()` method shown in the example below illustrates this.

Write the program

Okay -- you're ready to manipulate the snooze interval. You do this by adding get and set methods for the snooze interval. When you have an instance variable like `snoozeInterval`, you will regularly call the get and set methods `getSnoozeInterval()` and `setSnoozeInterval()`.

```
public class AlarmClock {
  long m_snoozeInterval = 5,000; // Snooze time in millisecond

  // Set method for m_snoozeInterval.
  public void setSnoozeInterval(long snoozeInterval) {
    m_snoozeInterval = snoozeInterval;
  }

  // Get method for m_snoozeInterval.
  // Note that you are returning a value of type long here.
  public long getSnoozeInterval() {
    // Here's the line that returns the value.
    return m_snoozeInterval;
  }

  public void snooze() {
    // You can still get to m_snoozeInterval in an AlarmClock method
    // because you are within the scope of the class.
    System.out.println("ZZZZZ for: " + m_snoozeInterval);
   }
}

public class AlarmClockTest {
  public static void main(String[] args) {
    // Create two clocks. Each has its own m_snoozeInterval.
    AlarmClock aClock1 = new AlarmClock();
    AlarmClock aClock2 = new AlarmClock();

    // Change aClock2. You use the set method.
    aClock2.setSnoozeInterval(10,000);

    aClock1.snooze(); // Snooze with aClock1's interval.
    aClock2.snooze(); // Snooze with aClock2's interval.
  }
}
```

Defined now are two methods to manipulate the snooze interval. One is used to get the snooze interval, and the other is used to set it. That may seem trivial, but then, `AlarmClock` is a trivial class. In future columns, the class will grow in functionality and complexity.

Conclusion

You've covered a great deal of new ground. You looked at how to manipulate primitive types like `int` and `double`. You examined local variables, method parameters, and variable scope. You learned how to add data to classes using instance variables, and how that data is contained in each instance. Finally, you explored encapsulation and how it leads to better code.

Next time, you'll see some more of the control structures in Java, such as `if` and `while` statements, and learn how to enforce encapsulation with Java's *access modifiers*. You'll also study some of Java's built-in functionality, which can manipulate time, and use it to add more features to the `AlarmClock` class. To accomplish that, you'll delve more deeply into how one object uses another to complete its work. ■

About the author
Jacob Weintraub is founder and president of LearningPatterns.com (LPc). Jacob has been working in object technologies since 1989, and teaching Java since 1995. He authored LPc's *Java for Programmers*, as well as many of its advanced courses, such as those on OOAD and EJB.

Home | Mail this Story | Resources and Related Links

Advertisement: Support JavaWorld, click here!

SunWorld
SUBSCRIBE HERE!
IDG's Web-only magazine for the Unix professional

(c) Copyright 2000 ITworld.com, Inc., an IDG Communications company

Resources

- "Learn Java from the Ground Up," Jacob Weintraub (*JavaWorld*, March 31, 2000) -- Jacob's previous **Java 101** column:
  http://www.javaworld.com/javaworld/jw-03-2000/jw-0331-java101.html
- Learn the basics, seek advice, and get help in *JavaWorld*'s community discussion just for beginners:
  http://forums.itworld.com/webx?230@@.ee6b804!skip=274
- Learn more about variables and primitive types:
  http://gamelan.earthweb.com/javaprogramming/javanotes/c2/s2.html

Feedback: jweditors@javaworld.com
Technical difficulties: webmaster@javaworld.com
URL: http://www.javaworld.com/jw-07-2000/jw-0707-java101.html
Last modified: Wednesday, July 12, 2000