Lesson 4: Working With Objects

"Object-oriented programming is an exceptionally bad idea which could only have originated in California."

—Edsger Dijkstra, Dutch computer scientist

Introduction

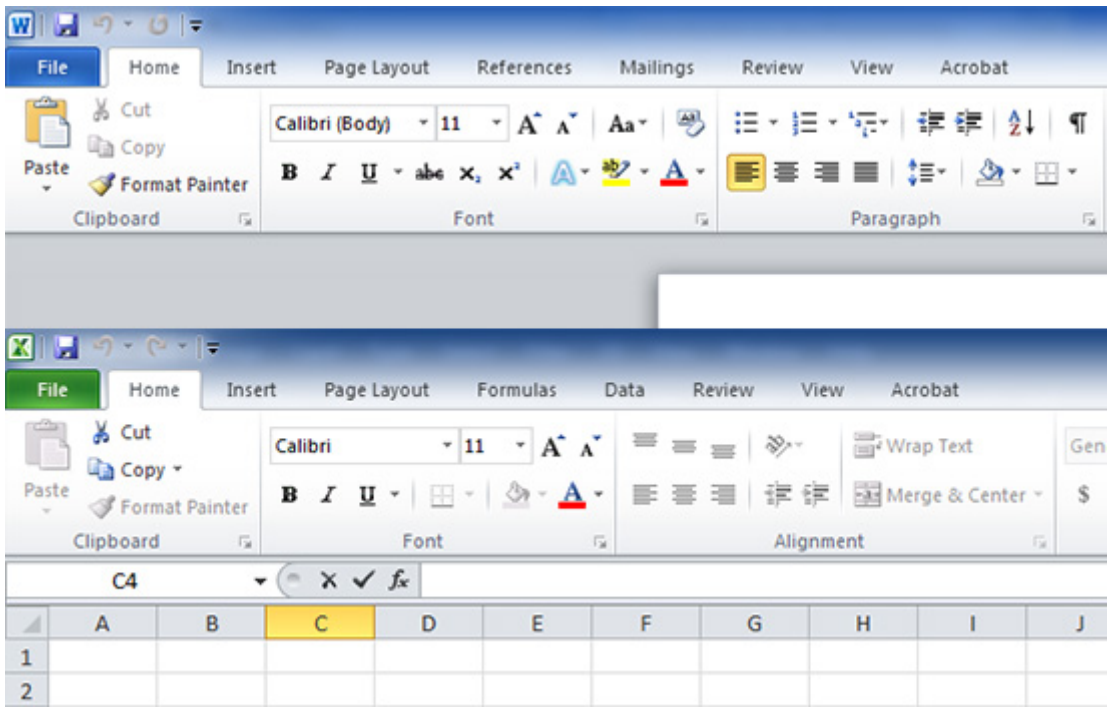
In this lesson, we'll look at how Java uses classes and objects. Java is an *object-oriented programming* (OOP) language. They're not as bad as Professor Dijkstra made them out to be!

OOP languages have two major benefits:

- **They're modular.** Years ago, programming was painstaking and solitary. One person generally wrote a program from beginning to end. But most of today's software is too complex for one person to build. OOP allows teams of programmers to crank out sophisticated programs in record time. Each person on the team is responsible for one or more small parts (*modules*) of the project. For a word-processing program, one programmer might work on the spell checker. Another might import art. A third might develop buttons for the toolbar. Using an OOP language like Java makes it easier to combine modules into a working whole.

- **They're reusable.** Have you noticed that Microsoft applications look alike? The toolbar (or "ribbon") and spell checker in Word look and act like the ones in Excel. Features from word processors crop up in spreadsheets and presentations. And the Save As dialog box looks the same in most applications.

OOP languages allow programmers to easily use modules in multiple applications. This can save tremendous amounts of time.



Because Java's an OOP language, it'll let you build programs around these reusable modules. And you don't have to hire a team of programmers. The folks at Oracle have given us hundreds of reusable modules to play with!

These modules come with the Java Development Kit (JDK) you downloaded. The JDK has a collection of reusable modules that OOP calls *classes*. The collection of classes in the JDK is the *Java Class Library*. One recent count I saw listed over 3,900 classes in this library.

You might be thinking, "What exactly are classes?" Good question! We'll get to that in the next chapter.

Java Classes: What They Can Do For You

A class is almost any type in Java other than one of its primitive types. I've mentioned String, a class that holds character strings, and we've discussed about Java's wrapper classes that enclose primitive types.

Another way to describe a class is a reusable module that you can use in your Java programs to save yourself time and hassle. Here are some examples:

- If you want to display text, the System class makes it easy using a default output device: your screen. We've used this class in our HelloWorld program and in other output statements in Lessons 2 and 3.
- A class named Color will help you manufacture all the colors you want.
- If you want to draw circles, rectangles, and lines, use the Graphics class to produce them.

You're getting the picture, right? Java's classes provide us with many capabilities we won't have to create ourselves.

My last part of the definition of a class is that it's a structure that contains information and actions that use the information. The information is the *attributes* of the class, or its data. The actions are its *methods*. (Some languages call methods "functions" or "subroutines.")

Creating Something From Nothing

To write programs, we'll need to write our own classes from scratch. So we'll start by looking at what goes into a class and what we get out of it.

A class is like a blueprint. It contains all the necessary information to build *objects* (just as a blueprint does for a building) and all the necessary actions those object can perform, such as opening windows or turning on air conditioning in a building.

A class is like a blueprint.

As you work on programs in this course, you'll build objects from classes to accomplish your goals. In the same way a contractor uses a blueprint to build a house (or several similar houses), you'll use a class to build one or more similar objects in your program.

Every time you use a class to create an object, you've created an *instance* of that class. (Instance is the formal OOP term; most programmers use the term *object* just as often. They mean the same thing.)

Houses built from a single blueprint have similar features. In the same way, objects built from the same class have similar attributes and capabilities, even though the details may be different.

Houses from a single blueprint

In the same way, if you use Java's JButton class to create two clickable buttons in one window, you create two different instances of the class. Both buttons would have the same characteristics (attributes), including text, size, location, and color. But the values in those attributes would be different. One might have the text "OK" while the other might say "Cancel." They could be different sizes and colors, and they should be in different locations.

You'd use the same actions (methods) to format the buttons, to set their colors, size, and location, and to change the text on each button.

Before we talk about how to use classes, let's discuss about their attributes and what's inside them.

What Are Class Attributes?

In order for most classes to be useful, they have to contain information, or data. Data elements in a class are called its *attributes*. I'll usually say *data* since that's a shorter and familiar term, but you'll hear the term *attribute* as well.

Since classes represent things (buttons, fonts, or windows, for example), attributes are data items that describe those things. JButton objects, for example, have attributes that describe color, position, size, and text, among other things.

When a programmer defines a class, he or she defines data elements that the class needs in order to work. And since the creator of a class knows how that class works, she or he is the best one to make sure that values in those attributes are valid and don't get changed by just anyone at any time. The creator of the class controls how the class changes and uses data values. So programmers usually hide attributes.

If the programmer has hidden the data, you might wonder how another program or class is going to use it. The answer is the methods of the class.

The author of a class is responsible for defining actions that are valid for the class. Some of those actions will allow other classes and programs to access the hidden data. And the actions of a class are defined in its methods.

What Are Methods?

Let's do a quick review. You know now that a Java program can use objects. You also know that each object gets built from a blueprint we call a class. Finally, you know that every object you create is an instance of that class, and that the object's hidden attributes describe it.

If you want your program to be useful, you're going to need to make your objects do something! You do that with methods . . . step-by-step instructions that tell objects what they need to do to be useful.

Suppose you want to write a Java program that displays a couple of buttons (or, if you want to get technical, instances of the JButton class). Let's further suppose you want to change the buttons' colors, text colors, sizes, and locations. Since you don't know the names of the attributes in the objects that control these buttons, how do you change them? By using the buttons' methods.

And how can you find out about the methods? Java has some documentation called its *API*, or application programmer interface. There's a link to the API in the Supplementary Material at the end of this lesson.

If you look up the JButton class in the API, you'll find methods called setBackground(), setForeground(), setLocation(), and setSize(), among others. These methods update the button's attributes for us. They also make sure that we don't use negative sizes, bad locations, or invisible colors. So methods set data values and protect attributes from errors. Since programmers want others to use methods, most methods are visible.

I've included a lot of terms and definitions in this chapter. So before we move on to Chapter 3, try this game to see how well you remember them.

Text equivalent start.

Instructions: Read the clue in the first column, and guess the term. Then read the second column for the answer.

clue	Answer
These include text, size, location, and color.	Attributes
These step-by-step instructions tell objects what to do.	Methods
This is the Java equivalent of a blueprint.	Class

Text equivalent stop.

Was the Hangman game difficult? It's hard to get a clear picture of how things like this work without actually seeing them in action and using them. So let's take an OOP approach to our HelloWorld project.

Creating an OOP Version of HelloWorld

Creating an OOP version of HelloWorld may seem unreasonably complicated and complex. For simple Java projects like HelloWorld, it's much easier to write a short main() method to display text.

As soon as you get into larger projects, though, doing everything in one method becomes impossible. So we're going to write a simple project in an OOP form. As always, feel free to share questions with other students in the Discussion Area.

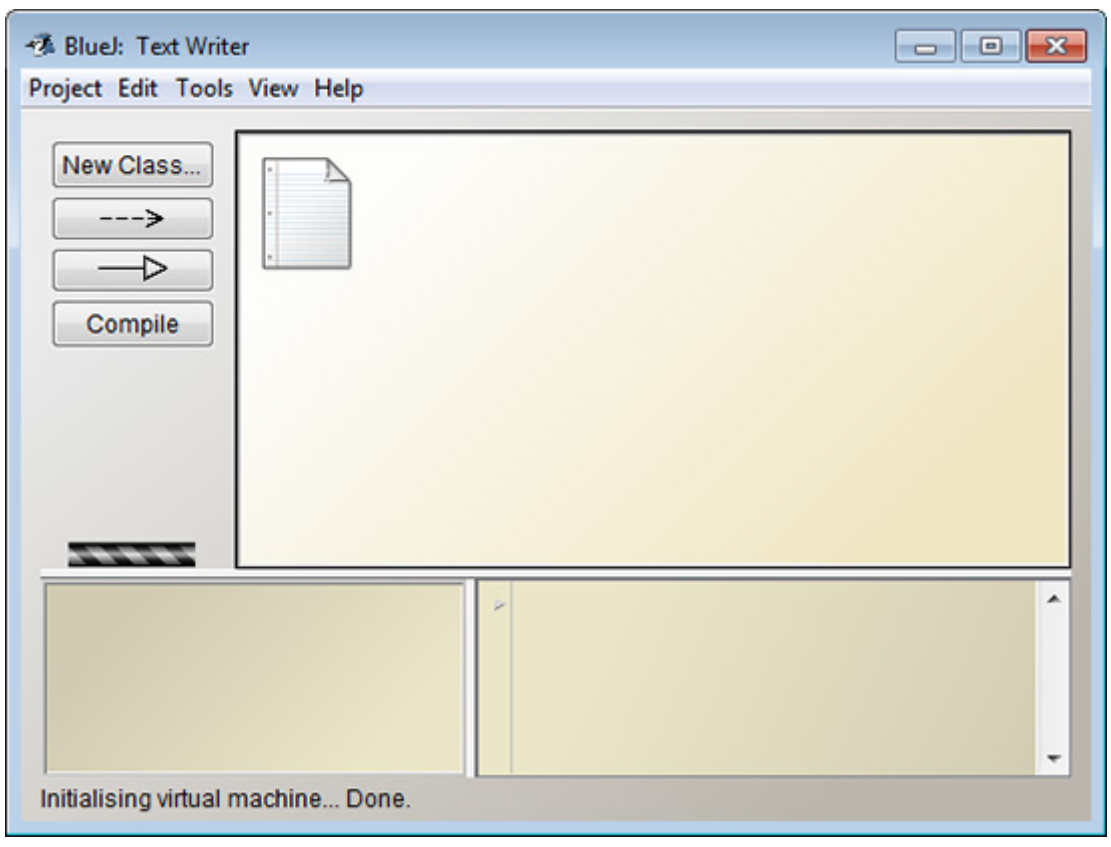
Now on with the fun and games!

Designing a Class

We want to design a class that's useful and reusable. All our HelloWorld program did was display the text "Hello, world!" We can't change its output unless we rewrite, recompile, and rerun it. I'd like our OOP approach to be more flexible, so let's design a class that will display any text we give it. Let's make it store the text internally, so the text can display more than once if we want it to.

Someone who used this class to display text would create an object from our class, tell it what text to display, and then ask it to display the text.

First start a new BlueJ project, just as you did for HelloWorld in Lesson 2. Choose the **New Project** option from the Project menu, and name it anything that makes sense to you; I named mine Text Writer. Once you've created your project, you'll have a BlueJ project window without any class icons in it. Here's a screenshot of what that might look like:

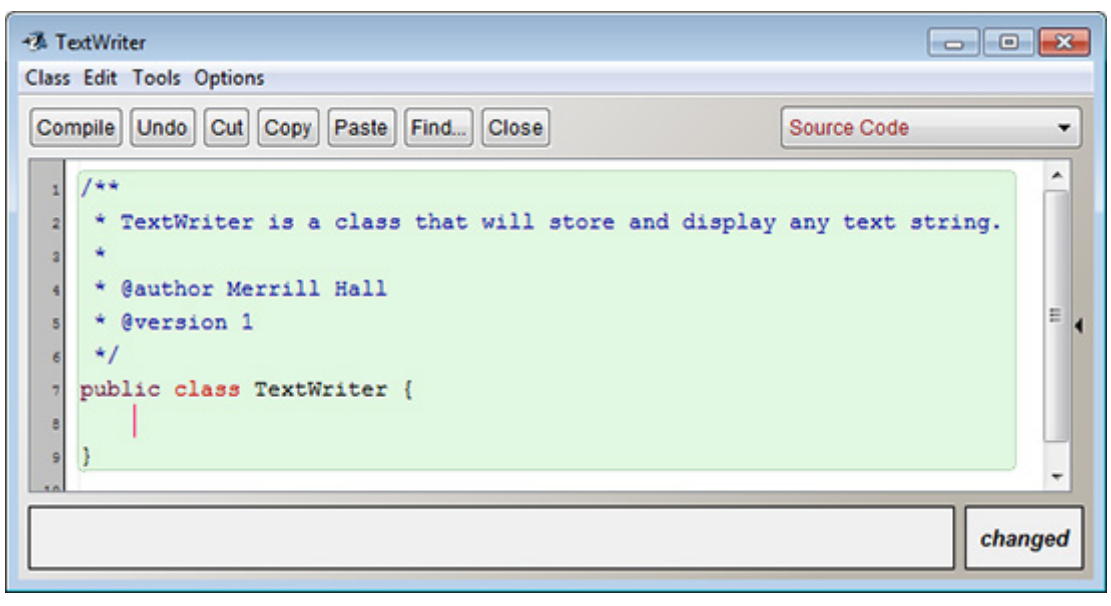


Text Writer project

Now create your new class. I named mine TextWriter. Java names can't have any spaces in them, so I ran the words together.

When you open the editor window for the class, you'll see the code that BlueJ generated. I'll leave it to you to update comments as you like, but remember that it's a good idea to keep them up to date.

Once you have the editor open, delete all the code in the class so you have an empty class, like this:



TextWriter class declaration

I'm not going to show the whole window all the time; most of the time, I'll just show you the code, both to save space and so you can copy and paste it.

Now let's write our class. We need attributes (data to work with) and methods to do the work.

Let's start with attributes. What information do we need in our class? All we'll need for this class is someplace to keep the text we want to display.

Declaring Instance Data

In Java, before we can save data in an attribute or get data out of it, we have to tell Java about it. We call that *declaring* the attribute. We'll often use these two types of data elements: *instance variables* and *local variables*. We already saw some local variables in Lesson 3, in the main() method.

An *instance variable* is a little different. It's a variable (or data attribute) that will exist in each and every instance (or object) of the class. That's how we'll want to store text in TextWriter, so if we create three TextWriter objects, each object will have its own variable and be able to store its own text.

Declaring an instance variable is similar to declaring a local variable. We need to tell Java its type and its name.

But an instance variable declaration also differs from a local declaration. You declare instance variables inside the class but outside the methods. Traditionally, instance variable declarations come first in a class, so we can see right at the top of the class what its data elements are.

Instance declarations also have an additional modifier. They take this form:

<access modifier> <data type> <variable name>;

Text equivalent start.

If the access modifier is public, then the variable name is accessible by other classes, and the variable name can contain any data type including: byte, short, int, long, float, double, boolean, char, or String.

If the access modifier is private, then the variable name is not accessible by other classes. The variable name for a private modifier can contain any data type including: byte, short, int, long, float, double, boolean, char, or String.

Text equivalent stop.

An *access modifier* comes first, and it tells Java who can see and use a variable. The two most commonly used access modifiers in Java are public and private. We already saw public when we defined HelloWorld class and its main() method. A public modifier tells Java that any program or class can see and use the name.

We just said in Chapter 2, though, that we want our attributes to be hidden. The private modifier does that. It tells Java to allow users to see and use the variable inside this class only.

After the access modifier, the variable's type and name appear just as they would in a local declaration. As you learned earlier, it's possible to store text in String objects, so that will be our type. Since this variable will hold text we want to display, I'll call it textToDisplay.

The declaration should look like this:

private String textToDisplay; // the text to be displayed

Once that's in our class, it'll look like this. (I'm not going to repeat all the comment lines in the interest of saving space.)

```
public class TextWriter {

    private String textToDisplay; // the text to be displayed

}
```

In Lesson 3, we discussed about initializing variables (giving each one a value that you may or may not change later). If you don't initialize instance variables, they have default values.

- For numeric instance variables, the default is zero.
- For Booleans, the default is false.
- For characters, it's the null character (which isn't printable).
- For reference variables like our String variable here, it's null, which means the variable has no object to refer to.

Since I didn't initialize our variable in the declaration, it's a null reference. There's no string for it to refer to yet. We'll get a string for it later in the lesson.

Designing Methods

Now that we've declared a data attribute for our class, we can move on to its methods, which define the actions our objects will be able to perform.

When someone wants to run one of the actions, we say that person will *call the method* that does the action. When a program calls a method, the computer goes to that method, runs its action or actions, and then comes back (returns) to the place in the program that it was called from. We'll see the details of how to do that in the next chapter.

In our TextWriter class, I'm going to define three methods. Then I'm going to talk about *constructors*—special methods that initialize objects.

The actions I want to include in TextWriter are:

- Storing a string value to be displayed

- Getting a stored value
- Displaying a stored value

OOP divides methods into two categories. I'm not too fond of the category names, but I didn't get to choose them. The categories are *mutators* (methods that change the value in a variable) and *accessors* (methods that just look at values without changing them). The first method I described in the bulleted list is a mutator, and the other two are accessors.

Now that we've got the terminology out of the way, we can get on with the design.

Since we need to store a value before we can display it, we'll do the update method first. In Java, methods that set values are often called *set methods* or just *setters*, terms I prefer to "mutator." Programmers usually name them "set" and then the name of the variable. So I'll name this method setTextToDisplay().

Let me show you the method, and then I'll explain its parts.

```
public void setTextToDisplay(String inputText) {
    textToDisplay = inputText;
}
```

We already know what public means—that anyone can call this method. The next word tells Java what type of value (if any) this method should return to anyone who calls it: The word void tells you that this method won't return any value.

Next comes the method name, setTextToDisplay. Then there are parentheses. A name with parentheses after it is always a method name.

Inside the parentheses, we tell Java what information we expect from whoever calls the method. The formal name is a *parameter list*. Each parameter has a type and a name. The list can have as many parameters as we need, separated by commas.

We need only one piece of information for this method, and that's the text we want to store. So the type for our only parameter is String.

Besides a type, we also need to give Java a name for each parameter. In this case, I named the parameter inputText. I said we should use descriptive names, and I'm trying to do that.

That describes the first line of our method. We call that line the method's *signature*; it tells Java the method's name, what information it needs, and what information it returns. That's all a user needs to know in order to call it.

Following the signature, we always use a left curly bracket to start the body of the method. In this case, the body is simple. It takes the text the user gives us in the parameter inputText and stores it in the instance variable textToDisplay. We'll use an assignment statement like those from Lesson 3:

textToDisplay = inputText;

It copies the reference (or memory address) of the text string from inputText into textToDisplay, so it's no longer a null reference. Now it has a string to point to.

That's all our method needs to do, so we close the method with a right curly bracket:

```
public class TextWriter {
    private String textToDisplay; // the text to be displayed

    public void setTextToDisplay(String inputText) {
        textToDisplay = inputText;
    }
}
```

The second method, which will give callers a copy of the stored string, is a *get method* or a *getter*. Here's the method:

```
public String getTextToDisplay() {
    return textToDisplay;
}
```

The second word, String, tells Java that whoever calls this method will get a text string back from it. There's no parameter list—just empty parentheses. This method doesn't need any information from its callers; they're asking us for something instead.

The information we want to return is in the variable textToDisplay. That's what the return command does—it tells Java what to return to the caller.

The third method displays the text, so I'll name it display. Since we've seen an output statement in our first version of HelloWorld, this method shouldn't need any explanation:

```
public void display() {
    System.out.println(textToDisplay);
}
```

With all three methods in it, our class looks like this:

```
public class TextWriter {
    private String textToDisplay; // the text to be displayed

    public void setTextToDisplay(String inputText) {
        textToDisplay = inputText;
    }

    public String getTextToDisplay() {
        return textToDisplay;
    }

    public void display() {
        System.out.println(textToDisplay);
    }
}
```

Let's move on to the next piece of the puzzle.

Writing Constructors

Constructors have a special purpose: to initialize new objects. We use constructors whenever we create new objects. If we write a class without giving it a constructor, Java will supply a default constructor that initializes instance variables to default values.

Some Java professionals argue that constructors aren't methods. But they're so similar to methods that I consider them methods with a special purpose. There's more information about this topic in the Supplementary Material if you're interested.

I always write at least one constructor. First, that way I can tell Java what should go into each instance variable, even if it's the same value Java would put there. Second, it's good documentation for anyone else who reads my code; they'll see what to expect when they create a new object.

Constructors look a bit different from other methods, though. First of all, the constructor's name is always the same as the class name. And a constructor doesn't have a return type—not even void.

Let me show you two constructors I'm proposing for our TextWriter class, and then I'll discuss about them a bit more before we wrap up this lesson.

```
public TextWriter() {
    textToDisplay = "";
}

public TextWriter(String inputText) {
    textToDisplay = inputText;
}
```

Please add these constructors to the top of the TextWriter class, between

```
public class TextWriter {
```

```
   private String textToDisplay;
```

and

public void setTextToDisplay(String inputText) {

That's where most programmers put constructors—after instance variables and before other methods. Java doesn't care where they are in the class as long as they're not inside other methods, but putting them here makes them easy to find.

You can see that the two constructors have the same name. How does that work? Well, it's an example of what we call *overloading*. In Java I can overload methods and constructors, which means having two or more methods with the same name. Java tells them apart using their parameter lists, which have to be unique either in the number of parameters, the types of parameters, or both.

The last time I looked, the String class had 13 different constructors. They allow us to build strings from a lot of different sources.

In the case of the constructors above, the first one (called the *default constructor* since it has no parameters) will initialize my object with an empty string. The second constructor will initialize the object with whatever string is in its parameter.

It's up to the designer of a class to decide the best way to create objects, what information has to be there (if any), and how many constructors to include.

On the other side of the issue, when I want to create an object from a class that has more than one constructor, which one I use is up to me. In the case of the TextWriter class, I can call its default constructor to create an object with an empty string and add the text later using setTextToDisplay(). Or I can create the object with text already in it by using the second constructor.

How do we call constructors, anyway? And since constructors can run only when we create objects, how do we create objects? Read on!

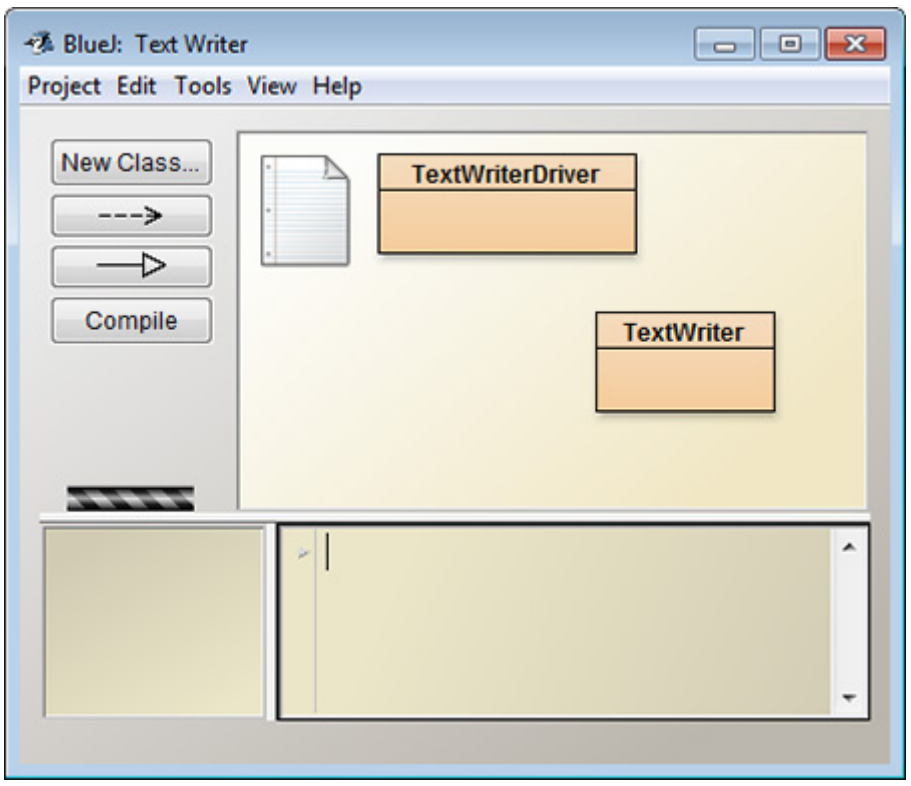
Creating Objects

We've built a class that can create objects, and these objects can store a text string using either the constructor or the setTextToDisplay() method. The objects can also display the text string using the display() method, and they can give the text string back to the caller using the getTextToDisplay() method. But we still need to know how to tell the class we want to create objects and call their methods.

Let's go back to BlueJ's main project window. You can close the editor window for TextWriter now if you like; we won't need it for a few minutes.

In BlueJ's main window, click the **New Class** button, and create another class named TextWriterDriver. We'll put code in this class that will create and use TextWriter objects. Programmers often do this to test classes they've written: They create another class whose only purpose is to use and test the first class. Classes of this second type are *driver classes*.

A Create New Class dialog box will appear. Click **OK**, and a second icon should appear in the BlueJ window with the new name in it, something like this:



BlueJ project window with multiple classes

Your icons may be arranged differently, but you can drag them around anywhere you like.

A project can include any number of classes. The largest BlueJ project I have on my computer right now has 63 classes in it. We won't need any large projects like that, but I just want you to know that you can have more than one class in a project.

Now that you've created the class, open its editor window, and delete the code that BlueJ provided. Replace it with this code:

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


    }
}
```

This will be an executable class since it has a main() method. And that's the only method it will have, since its only purpose is to try out our TextWriter class.

To test everything, I'll need to try both constructors and all three methods. I'll also show you a little more about declaring variables. First I'll add this variable declaration inside the main() method:

```
TextWriter tw1;
```

This tells Java I want a variable in my main() method: a variable named tw1 whose type is TextWriter. I named it that because it's going to be my first TextWriter variable; I'll add another one in a little bit. This will be a local variable in main(). Since the type is TextWriter, Java knows it'll refer to a TextWriter object.

Since we want the variable to refer to an object, let's create one for it. Add this line after the last one:

```
tw1 = new TextWriter();
```

We already discussed about assignments, and that's what this statement is. But what does it assign to tw1?

The right half of the statement tells Java to create a new TextWriter object. (The new keyword always creates a new object.) The class name, TextWriter, tells Java what type of object to create. And since I didn't put anything in the parentheses after it, Java knows to call the default constructor. So after that statement runs, I'll have an object to try out.

Next, let's store some text in the object. We'll call the object's setTextToDisplay() method to do that. When we call an object's method in Java, we tell Java what object to go to, and then which of its methods to run, like this:

```
tw1.setTextToDisplay("This line is from my first Java object!");
```

The period between the object name and the method name is Java's *dot operator*. As I mentioned in Lesson 3, programmers use the dot operator to tell Java that the name after the dot is inside the name before the dot. In this case, the method setTextToDisplay() is part of the object that tw1 refers to. I've given the method a text string argument that should get stored in the object's textToDisplay variable.

How can I make sure it got stored there? How about by displaying the contents of the variable? I can do that by calling the object's display() method, like this:

```
tw1.display();
```

The display() method doesn't accept any arguments, so I didn't enter any.

Okay! That's enough to try out. If you run your main() method, it should produce this output:

Text equivalent start.

BlueJ Terminal window showing the following output:

```
This line is from my first Java object!
```

Text equivalent stop.

If it doesn't, come over to the Discussion Area,and post your queries.

So far I've tested one constructor and two methods. Let's wrap this up with two more lines of code:

```
TextWriter tw2 = new TextWriter("And this line is from my second object.");

System.out.println(tw2.getTextToDisplay());
```

The first line does three things:

- The left side of the assignment declares a new TextWriter variable named tw2.
- The right side creates a new TextWriter object using the second constructor, since I put a string argument into its parentheses.
- The assignment connects the variable to the object.

I still have the getTextToDisplay() method to try. The last line of code above calls that method for object tw2 to retrieve its text, and whatever text it gives back will be displayed by the call to println(). When you run the driver program, you should see this:

Text equivalent start.

BlueJ Terminal window showing the following output:

```
This line is from my first Java object!
And this line is from my second object.
```

Text equivalent stop.

Please join me in Chapter 5 for a summary of classes, objects, and driver programs.

Summary

In this lesson, we've written our first class that's designed to create objects, and we've written a driver program that tests it. Along the way, you learned about instance variables, methods, constructors, parameters, and creating and using objects.

I was in a fog for several lessons in my first programming class, so I just copied and edited the teacher's code. But once it sank in, I found out I really enjoyed it . . . and I still do almost 40 years later.

Be sure to do the assignment, take your quiz, and feel free to drop by this lesson's Discussion Area and share your thoughts with other students.

And in case it helps, here are the two complete classes we built in this lesson so you can compare your code to mine if you'd like. Again, I've left out comments in the interest of space.

TextWriter Class

```
public class TextWriter {
    private String textToDisplay;

    public TextWriter() {
        textToDisplay = "";
    }

    public TextWriter(String inputText) {
        textToDisplay = inputText;
    }

    public void setTextToDisplay(String inputText) {
        textToDisplay = inputText;
    }

    public String getTextToDisplay() {
        return textToDisplay;
    }

    public void display() {
        System.out.println(textToDisplay);
    }
}
```

TextWriterDriver Class

```
public class TextWriterDriver {
    public static void main(String[] args) {
        TextWriter tw1;
        tw1 = new TextWriter();
        tw1.setTextToDisplay("This line is from my first Java object!");
        tw1.display();
        TextWriter tw2 = new TextWriter("And this line is from my second object.");
        System.out.println(tw2.getTextToDisplay());
    }
}
```