Chapter 5 Towards Consistent Classes

“When facing a tree, if you look at a single one of its red leaves, you will not see all the others. When the eye is not set on any one leaf, and you face the tree with nothing at all in mind, any number of leaves are visible to the eye ...”

Takuan

Introduction

Takuan was a Buddhist master who lived in the middle of the second millennium. You may wonder how a 500 year old quote from a Buddhist is relevant to computing; let me explain. Buddhists had been studying human nature for 2000 years at that point; and people haven’t changed much in 500 years. Our culture (language, technology, education) has changed radically since, but our DNA has not; our bodies, our brains, our minds are much the same. Buddhist philosophers recommend paying complete attention to whatever you are doing, whether it is writing code or washing the dishes (see Thich Nhat Hanh, The Miracle of Mindfulness). The resulting focus can change your life.

How is object programming different from procedural programming (from which it grew)? In this context, the leaves on Takuan’s imaginary tree are classes. If you are practicing object programming correctly, when you are writing a class, you are thinking of nothing but that class. This provides tremendous power, because you are not distracted and can focus on writing a bug-free class. Once it is written and tested, you can forget what’s inside and devote your attention to other matters. This is one of the most important advantages of object programming.

A major constraint on a programmer’s ability to think clearly (and thus solve the problems that inevitably arise when programming) is cognitive overhead. If every class is similar, once the pattern becomes familiar, the programmer’s cognitive overhead is reduced. This chapter will present a method for writing classes that will be used in every class from now on. There are many styles of programming, and no one can say that one is best for everyone. The style recommended here is simple and consistent; that’s enough for now.

This chapter includes more complete explanations of the components of classes. That requires a fair amount of detail and a bit of notation. So, sections presenting some of the associated details will bracket the sections describing class components. Read those detail sections briefly, but return to them in a few days; the details are a trifle dull, but learning them will save you many hours later.

Details I -- Statements in Java: Syntax and Semantics

So far, the code in the book has only used three kinds of Java statements: assignment statements, message statements and return statements. They can be easily distinguished. Every assignment statement has an equals sign, like, balance = 17;. Every return statement starts with the word
return. Every message statement has a ".", like, System.out.println("greetings");, although sometimes the "."s are added automatically by the compiler and do not appear in the source code. The good news is there are only about a dozen statements total in Java. You may be wondering what all the rest of the code so far has been composed of; mostly declarations (class, method, and variable). These will be covered in the next section.

Syntax and Semantics

Syntax is another word for grammar. It has nothing to do with meaning. The syntax of Java (or any programming language to date) is very simple compared to a natural language like English. The compiler uses what is called a context free grammar to check the syntax and, if it is correct, then converts the source code to byte code (See “The Java Virtual Machine” in Chapter 1). A context free grammar is composed of a set of productions. The syntax of each statement is defined by a single production, as will be seen below. Once you have learned the syntax of each of the elements of Java, the mystery of syntax errors will be mostly dispelled.

Semantics is another word for meaning. In the context of programming, semantics means action (this is “operational” semantics). The semantics of a statement is the action it performs when it is executed.

BNF Notation

The productions of a grammar are often represented in BNF notation, which stands for Backus Naur Form (Backus and Naur were the originators of this representation of grammars). This is a metalanguage, i.e. it is a language which is about a language. Since you are not familiar with the Java language, and you are familiar with English, here’s an introduction to BNF for a subset of English.

BNF For a Tiny Fragment of English

Consider a fragment of English where every sentence is composed of a noun phrase followed by a verb phrase. This could be written as the BNF production:

<sentence> ::= <noun phrase> <verb phrase>

The “::=” means “is defined as”. So, this is read, “A <sentence> is defined as a <noun phrase> followed by a <verb phrase>”. Things between pointed brackets are called non-terminal symbols and must be defined somewhere in the grammar. So to complete this grammar we would need definitions of <noun phrase> and <verb phrase>. For now let’s say that the only legal verb phrases are “runs” and “jumps”. In BNF:

<verb phrase> ::= runs | jumps

The vertical bar () means “or”. Symbols without brackets, like “runs” and “jumps” are called terminal symbols and must appear literally.

Let’s say that a noun phrase is either a proper name, or an article followed by a noun. In BNF:
Let the proper nouns be “Jane”, “Dick”, or “Spot”, the articles, “a” or “the”, and the nouns be “cat”, or “mouse”. In BNF:

```
<noun phrase> ::= <proper noun> | <article> <noun>
<proper noun> ::= Jane | Dick | Spot
<article> ::= A | The
<noun> ::= cat | mouse
```

All the non-terminals have been defined (i.e. they have appeared on the left of a ::=) so the grammar is finished. What language does it generate? In other words, what are all the legal sentences in this language? Here they are, in left to right order:

Jane runs
Jane jumps
Dick runs
Dick jumps
Spot runs
Spot jumps
A cat runs
A cat jumps
A mouse runs
A mouse jumps
The cat runs
The cat jumps
The mouse runs
The mouse jumps

So, there are just 14 legal sentences in the language. Let’s add optional adjectives to make it a bit more interesting.

**Adding Adjectives**

The notation for an optional symbol is [optional thing]. So, to allow the two sentences: “The big cat runs”, and “The cat runs”, the following productions will do:

```
<noun phrase> ::= <proper noun> | <article> [<adjective>] <noun>
<adjective> ::= big | small | black | white | ferocious
```

English allows multiple adjectives to modify a noun. If we wanted to allow a sentence like, “The big black ferocious cat jumps”, the requisite production would be:

```
<noun phrase> ::= <proper noun> | <article> [<adjective>]* <noun>
```

Notice the “*” after the []’s. This means “0 or more repetitions of the symbol in the []’s”.

Here is the grammar all collected together:
BNF, Java and Adaptive Systems

The Java compiler, like any contemporary compiler, is very literal and rigid. It insists that source code match its grammar, symbol by symbol. Any deviation will result in compiler errors; and errors prevent the compiler from producing byte code, and without byte code you can’t execute your program.

Syntax errors are just details, but they are details that can stop you. If you omit, or misuse a comma in an English paper, you may be corrected by your teacher, or lose points, but your reader can still understand your meaning. The compiler, by contrast, will not. If a program is missing a semicolon, or has a misspelled word, no matter how many times you compile it, it will never run. In the person/compiler system, the person is the adaptive system; if a program does not compile, the person must make the adjustment, the compiler will not. Fortunately, once you know the BNF, you will know what the compiler is looking for (usually!).

The Assignment Statement

The assignment statement, while simple and unprepossessing is the only one that changes the state of a program. Its grammar is shown in BNF 5.1.

BNF 5.1 The Assignment Statement

<table>
<thead>
<tr>
<th>assignment stmt</th>
<th>::=</th>
<th>variable = expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semantics</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1. Evaluate the</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. Assign the value to the</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Every assignment statement matches this syntax. I.e. every assignment statement consists of a variable, followed by the assignment operator, followed by an expression, and finally a
semicolon. This has several implications: 1) The only thing that can appear on the left of the assignment operator is a variable and whenever you see an assignment operator, you know that whatever is to the left of it is a variable, otherwise it won’t compile into working code. 2) Only an expression can appear to the right of the assignment operator and anything that appears there must be an expression, for the same reason. 3) Any other syntax is illegal.

What is not specified by the BNF, but is necessary for an error free program, is that the type of the expression must be compatible with the type of the variable. If the types are not compatible a compiler error will occur. The details of compatibility appear in “Expressions” below.

Check which symbols in the following examples correspond to which BNF symbols. Examples:

```
balance = nuBalance;                  // see Listing 3.2
balance = balance - amountToWithdraw; // see Listing 3.4
currentAccount = account3;            // see Listing 3.11
myColor = c;                          // see Listing 4.11
```

**The Message Statement**

Computing is information processing (remember?). Almost all information processing in a Java program is accomplished by sending messages to objects. As BNF 5.2 shows, every message statement is an object followed by a period, followed by a message (which has parameters enclosed in parentheses).

**BNF 5.2 The Message Statement**

```
<message stmt> ::= <object>..<message>([<actual parameters>]);
```

**Semantics**

1. Perform the parameter linkage (See “Parameters (actual, formal, linkage)” below).
2. Execute the method body of the associated method, using the <object> as “this”.

The BNF does not specify that that method must be defined in the class the object belongs to (or one of its superclasses). Again, the compiler will catch the error if it is not (see “The mechanics of message sending” below).

Examples:

```
myAccount.setBalance(1234);       // from Listing 3.3
System.out.println("Greetings"); // from Listing 2.2
rightEye.growPupil();             // from Listing 4.15
```

But, here are some message statements that do not appear to match that syntax; they have nothing to match <object>.

```
initComponents();                  // from Listing 4.9
repaint();                        // from Listing 4.15
```

This is because these are shorthand for this.initComponents() and this.repaint(); The compiler fills in this for you (see “Special to Java -- What is This?” below).
**How to Generate a NullPointerException**

One of the most common run-time errors is the NullPointerException. That name is a bit worrisome, but actually descriptive. One way to generate one is to send a message to an object which has been auto-initialized to null. Recall that Java auto-initializes instance variables to zero, which, for references (and all Objects in Java are references), is called null. So, if you send a message, any message, to an object variable before you set it to reference an object, it will always generate a NullPointerException. Like this:

```java
1    public class Broke {
2       Object anObject;
3
4        void brokeMethod() {
5            theObject.toString();  // this will be a NullPointerException
6        }
7    } // Broke
```

**Line 3:** Declares an instance variable named `anObject`. This will be auto-initialized to `null` when a `Broke` object is created.

**Line 4:** When a `Broke` object is sent a `brokeMethod()` message, it will attempt to send `toString()` to `anObject`, but since `anObject` is still `null`, that is equivalent to `null.toString()`, which will cause a `NullPointerException`.

Every time you have a NullPointerException, it means your program has tried to send a message to an object that hasn’t been initialized (or has been set accidentally to `null`). Every time. No exceptions! Thus null pointer exceptions are easy to fix; just go to the line the exception was thrown and figure out why the object on that line in null.

See the Appendix B, “Exceptions”, for more details.

**The Return Statement**

Return statements are used to exit a method and can be used to pass information back to the point that the message was sent. If the return type of the method is not void, the compiler will insist that the `<expression>` exist and be of a type compatible with the type of the method.

**BNF 5.3 The Return Statement**

| `<return stmt>` ::= return `[<expression>]`; |
| **Semantics** |
| If there is no `<expression>`, return immediately to where the method was invoked. |
| If there is an `<expression>`: |
| 1. Evaluate the `<expression>` |
| 2. Leave the method immediately, returning that value as the value of the message that invoked the method. |
Accessors that get values have exactly one line in their bodies, a return statement. As you can see in Listing 5.1, the type int appears before getBalance() -- this is the type of the method.

Examples:
```
return balance;                      // from Listing 3.2
return currentAccount.getBalance();  // from Listing 3.11
```

Notice in the second example that the <expression> is a message; its value is whatever its method returns.

Now that you understand BNF notation and the syntax and semantics of the only three statements that actually accomplish anything, you are ready to learn the basics of classes.

**The Basics of Classes**

As discussed previously, a class declaration defines a template for objects (or instances) of that type. It includes both variable and method declarations. Variables contain information. Methods perform actions; what an object can do depends on what methods are declared and how they are implemented. Now that you have some experience with a few classes, it is possible to gain a more detailed understanding of how they work. A class may have many methods, but this section will only cover the standard methods, accessors and toString() -- these will be described more or less completely.

**Variables I (state)**

Variables store information; the state of an object is determined by the value of its variables. Every variable has a name, a type, and a value. When a variable is declared, only the name and type are required. There are variables of various sorts in Java, including: instance, class, local, parameter and method variables. In this section only instance variables will be presented. Their syntax is shown in BNF 5.4. This is a simplified BNF for a variable declaration; BNF 5.5 is a more complete definition.

**BNF 5.4 Variable Declaration 1**

```
<variable declaration> ::= <type> <identifier>;
```

<table>
<thead>
<tr>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create a variable of the given &lt;type&gt; with the name &lt;identifier&gt;</td>
</tr>
</tbody>
</table>

Examples:
```
public class Account {  // from Listing 3.1
    int balance;
    String name;
}
```

The Account class from Chapter 3 has been modified in Listing 5.1 to set the initial balance for every account to $1,000,000 and to set the initial value of name to "nobody" (Changes are in
If you do this and then later see an account with the name "nobody", you will know you forgot to initialize the name variable to something besides the default.

Listing 5.1 The Account Class With Auto-initialization

```java
public class Account {
    protected String name = "nobody";
    protected int balance = 1000000;

    Account(){   //empty default constructor
    }

    public int getBalance() {return balance;}

    public void setBalance(int nuBalance) {balance = nuBalance;}

    public void withdraw(int amountToWithdraw) {
        balance = balance - amountToWithdraw;
    }
}
```

Each Account object must keep track of the name and balance in a particular bank account. So, the Account class has two variables, balance and name; they must be declared outside of any method. The type of balance is int, the type of name is String.

The alert reader will have noticed that the two variable declarations in Listing 5.1 do not conform to the syntax in BNF 5.4. The protected keyword and the assignment are optional and do not appear in that BNF definition. A more complete BNF for a variable declaration is shown in BNF 5.5.

BNF 5.5 Variable Declaration 2

```
<variable decl>::= [<access>] <type> <identifier> [=<expression>];
```

<table>
<thead>
<tr>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Create a variable of type &lt;type&gt; with the name &lt;identifier&gt; with the &lt;access&gt; defined.</td>
</tr>
<tr>
<td>2. If the optional =&lt;expression&gt; is there, perform the assignment statement.</td>
</tr>
</tbody>
</table>

In the case of instance variables, the variable is created when the object is instantiated; i.e. in Listing 5.1, the balance and name variables will be created and initialized whenever new Account() is executed. If the optional [=<expression>] is there, then the variable is initialized to the value of the <expression>, otherwise to zero.

You have seen numerous examples of instance variable declarations that included initialization. In Listing 3.13 (copied below), theBank appears outside of any method in the ATM_Applet class; it is thus an instance variable. When the Applet instance is created (if you wish to convince yourself it is an instance, run any Applet and chose Clone from the menu to create another instance) its instance variables are created and initialized.
When the \texttt{new Bank()} constructor is executed, creating a new \texttt{Bank} instance, the four instance variables in the \texttt{Bank} class are created and initialized. Thus, Line 2, above, causes the four instance variables in Listing 3.10 (copied below) to be created.

```java
1 public class Bank {
2     Account account1 = new Account();
3     Account account2 = new Account();
4     Account account3 = new Account();
5     Account currentAccount = account1;   // to start with
```

All four create \texttt{Account} variables; the first three instantiate \texttt{Account}, so they each create a \texttt{name} and \texttt{balance} variable along with the instance (Lines 2 and 3 in Listing 5.1).

Two more examples are:
```java
String returnMe = "I am a Circle: "; // see Listing 4.3
protected Color myColor = new Color(100,0,100); // Listing 4.11
```

Each instance of a class has a copy of each \textit{instance variable}; that is why they are called \textit{instance} variables. The \texttt{balance} variable in \texttt{Account} is a good example; every account needs to keep track of a different balance, so every account needs its own \texttt{balance} variable to store that information.

**Methods (control)**

The standard methods include two accessors for each variable and a \texttt{toString()} method for debugging. Additionally, most classes have various constructors to make initialization simple, plus other methods which expand its capabilities. This section will present the syntax of method declarations, using examples from accessors you have already seen. Then a tool that writes these methods automatically will be introduced before the rest of the description of methods.

The syntax of every method declaration is a method heading followed by a method body as shown in BNF 5.6.

**BNF 5.6 Method Declaration**

```
<method decl> ::= <method heading> <method body>
```

**Semantics**

A method declaration is never executed. So it does not have semantics in the sense of statements. Nevertheless, it does have a meaning, namely: Create a method for the current class with the signature declared in the method heading. The body of the method is executed when the corresponding message is sent.

The method heading can take several forms since the access modifier, the return type, and the parameters are optional; the name, and parentheses are not optional; see BNF 5.7.
BNF 5.7 Method Heading

<method heading> ::= [<access>] [<return type>] <identifier> ([<formal parameters>])

**Semantics**

A method heading is never executed. It defines the signature of the method. Constructors do not have a return type. For ordinary methods, if the return type is not void, the method body must end with a return statement whose <expression> has a type compatible with the <return type>.

Examples (again, match the code symbols to the BNF):
void setBalance(int nuBalance) -- [<access>] omitted, no return, one int parameter
int getBalance() -- [<access>] omitted, returns an int value, no parameters
public String toString() -- access is public, returns a String, no parameters
public void paint(java.awt.Graphics g) -- access public, no return, one Graphics parameter.

The BNF for <identifier> and <formal parameters> are omitted (recall that an identifier is any series of letters, numbers and underscores beginning with a letter). Parameters are addressed below (“Formal and actual parameters”).

A method body is simply a block statement and a block statement is zero or more statements enclosed in `{}`; see BNF 5.8 and BNF 5.9.

BNF 5.8 Method Body

<method body> ::= <block statement>

**Semantics**

Same as the block statement.

BNF 5.9 Block Statement

<block statement> ::= { [<statement>]* }

**Semantics**

Execute each statement in the block in order.

Return Types

Every method that is not a constructor must have a return type. Any legal type may be a return type. if nothing is returned, the return type must be declared as void.

Accessors

Classes typically store information; other classes need to access that information, both to discover what it is and to update it. The methods that give other objects access to variables inside an object are called accessors. These are also called “getters” and “setters”, as they get and set the values of the variables they access.
Getters -- Get Values

Every getter is the same, except for three symbols. Compare the two getter methods in Listing 5.2; review Listing 3.2 if these seem unfamiliar.

Listing 5.2 Getters For Account

```java
1    public int getBalance() {
2        return balance;   // return the balance
3    }
4
5    public String getName() {
6        return name;   // return the name
7    }
```

The only differences are the return type (int or String), the name of the method (getBalance() or getName()), and the variable whose value is returned (balance or name). Both have a return statement as the only statement in the body of the method.

The int return type in getBalance() means that the getBalance() message is an expression of type int and so can appear anywhere an int expression is legal (See “Expressions” below).

Setters -- Set Values

Setters too are all the same shape. Compare the two in Listing 5.3.

Listing 5.3 Setters for Account

```java
1    public void setBalance(int nuBalance) {
2        balance = nuBalance; // set the balance
3    }
4
5    public void setName(String nuName) {
6        name = nuName; // set the name
7    }
```

Both are void (meaning they return nothing), with one parameter, which is used to set the variable involved. Both have a single assignment statement as the body of the method. They are identical in form, the differences stem from the names and types of the variables being set.

Parameters (actual, formal, linkage)

Parameters carry information into methods. The first step in executing a message statement is to perform the parameter linkage. To describe the parameter linkage requires a few new terms.

Formal and Actual Parameters

There are two varieties of parameters, actual parameters and formal parameters. Consider Listing 5.4, which is a copy of Listing 3.3 along with the setBalance() code from the Account class.

Listing 5.4 Formal and Actual Parameters

```java
1    class Account {
2        ...
```
public void setBalance(int nuBalance) {
    balance = nuBalance; // set the balance
}

// Account class
...

public static void main(String[] args) {
    Account myAccount = new Account();
    System.out.println("Before balance= " + myAccount.getBalance());
    myAccount.setBalance(1234);
    System.out.println("After balance= " + myAccount.getBalance());
}

The formal and actual parameters are shown in bold.

**Line 12:** sends the `setBalance()` message to the `myAccount` object with the parameter 1234.

**Line 3:** is the method heading for `setBalance()` and defines a parameter of type `int`. The former (1234) is the actual parameter, the latter (`int nuBalance`) is the formal parameter.

A mnemonic for which is which is that the formal parameter is part of the definition of the method and definitions are formal things. Plus, the actual parameter is the value that is actually sent along with the particular message that invokes the method (and different values may actually be sent with different invocations of the method!).

In BNF 5.10, the first production means, “<formal parameters> is defined as a <formal parameter> followed by any number of additional <formal parameter>s separated by commas”.

### BNF 5.10 Formal Parameters

<table>
<thead>
<tr>
<th><code>&lt;formal parameters&gt;</code> ::=</th>
<th><code>&lt;formal parameter&gt;</code> [, <code>&lt;formal parameter&gt;</code>]*</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;formal parameter&gt;</code> ::=</td>
<td><code>&lt;type&gt;</code> <code>&lt;identifier&gt;</code></td>
</tr>
</tbody>
</table>

**Semantics**
Declares one or more local variables which are created when the method is invoked and destroyed when it returns.

It may take a bit of thinking to realize why this BNF production generates that. The form is identical for actual parameters, as in BNF 5.11.

### BNF 5.11 Actual Parameters

<table>
<thead>
<tr>
<th><code>&lt;actual parameters&gt;</code> ::=</th>
<th><code>&lt;actual parameter&gt;</code> [, <code>&lt;actual parameter&gt;</code>]*</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;actual parameter&gt;</code> ::=</td>
<td><code>&lt;expression&gt;</code></td>
</tr>
</tbody>
</table>

**Semantics**
Each expression is evaluated before the message is sent as part of the parameter linkage.

When there is more than one parameter, the formal and actual parameters are matched up in the order that they appear. The first actual parameter is said to correspond with the first formal parameter; the second, with the second; and so on.
Each formal parameter is a variable declaration, i.e. a type and a name; each actual parameter is an expression of a type compatible (See “Expressions” below) with the corresponding actual parameter. Parameters of both types always appear between parentheses and are separated by commas.

**Parameter Linkage**

With the phrases “actual parameter”, “formal parameter” and “corresponding parameter” understood, the parameter linkage operation (the first step of the semantics of a message statement) may be expressed fairly succinctly. To perform a parameter linkage, for each parameter:

1. Evaluate the actual parameter.
2. Assign that value to the corresponding formal parameter. (Notice that this changes the value of the parameter variable, and thus, technically speaking, changes the state of the program. Thus the claim made previously that only assignment statements change the state of the computation is not quite true. But it is a convenient fiction.)

*What is the difference between this and the semantics of the assignment statement?*

Example: Listing 4.14 is the `shrinkPupil()` method for the `Eye` class. Its body is a single statement:

```java
pupil.setRadius(pupil.getRadius() - 2);
```

Answer these questions before reading the answers. It’s okay to look back to discover the answers if you know you know, but can’t remember.

*What type of statement is this?*
*What messages are sent?*
*What objects are they sent to?*
*What type is that object (i.e. what class is it an instance of)?*
*What is the actual parameter?*

*Describe the semantics of executing this statement.*

If you have no idea what most of the answers are, don’t feel bad, there is a tremendous amount of detail piling up here, and it’s all interrelated. Maybe it’s time to take a break and then, when your head is clear and you can focus again, reread the beginning of the chapter. You do want to learn this stuff, right?

```java
pupil.setRadius(pupil.getRadius() - 2);
```

Assuming you could answer most of them, here are the answers. This is a message statement. It sends the `setRadius()` message and the `getRadius()` message to the `pupil` object (which is a `FilledCircle`, See “The `Eye` class” in Chapter 4); the first with the actual parameter `pupil.getRadius() - 2`, the second with no parameter.

To execute that statement, Java first performs the parameter linkage, then executes the message body. The parameter linkage has two parts: First, evaluate the expression, second, assign the
value thus derived to the formal parameter. To evaluate a message statement, Java executes the inner message statement and uses what is returned as the value, so, the first thing that happens is pupil.getRadius() is executed.

The getRadius() method has no parameters, so the action is simply to execute the method body, which is one statement (return radius;); this sends back the value of the radius variable from inside the FilledCircle named pupil. Assume that its value is 50.

To complete the evaluation of pupil.getRadius() - 2 Java subtracts 2 from the 50 that came back from getRadius(), yielding 48. That is then assigned to the nuRadius parameter in the setRadius() method (finishing the parameter linkage). The body of that method is the assignment statement radius = nuRadius; whose execution stores the value of nuRadius (namely, 48) back in the radius variable inside the pupil object.

That’s a lot of detail! The good news is that you don’t actually have to think about that when you’re programming, once you understand the basics; it all disappears into expertise. You just think something like, “Hmmm, shrinkPupil() needs to make the pupil a little smaller. How about 2 pixels? Okay. pupil.setRadius(2 less than it is); Hmmm, how to find out what the radius is now? Oh yeah, getRadius(). So, pupil.setRadius(pupil.getRadius-2);." -- and type that.

**The toString() Method**

This is the standard debugging output routine for every class. Its signature, public String toString(), conveys several pieces of information to a Java savvy reader: 1) the name of the method is toString, 2) it has no parameters, and 3) it returns a value of type String. When writing a class, you can make toString() return any information you choose, so long as it is in a String. The most obvious information to include is the type of the object (i.e. what class it instantiates) and the current value of its variables.

Listing 5.5 is the toString() method from the Circle class. The body of the method has 5 statements.

Listing 5.5 The toString() Method for Circle

```java
1   public String toString() {
2       String returnMe = "I am a Circle: ";
3       returnMe += "\tx= " + getX();
4       returnMe += "\ty= " + getY();
5       returnMe += "\tradius= " + getRadius();
6       return returnMe;
7   } // toString()
```

The first, line 2, is a variable declaration statement with initialization to the String "I am a Circle: ". This variable, returnMe, is not an instance variable, because it is declared in the body of a method it is a method variable (See “Local Variables” below) The next three each concatenate: 1) a tab (\t), 2) an instance variable name, 3) an equals sign, and, 4) the value of
that variable, obtained through its accessor. Line 6, the last statement, is a `return` statement which sends back the value of `returnMe` as the value of the `toString()` message wherever it was sent.

**The ClassMaker Tool**

The process of programming is not simple, nor is it easy. Even after you understand the concepts, there is a host of details that must be attended to before a program can be completed. Many of the details a programmer must deal with stem from the rigidity of the software, which stems from the mindlessness of current computers.

On the other hand, computers are incredibly useful owing to their blinding speed and disregard for endless repetition. In this regard, with the appropriate software they are a pretty good complement to human skills/proclivities; tasks that always involve the exact same actions can be mechanized. Robot factories, good!

The large majority of Java programming time is spent selecting, designing, and implementing classes. Although the very simplest programs may have only one or a few classes, any substantial project has many classes. Once you have mastered accessors, constructors and `toString()`, writing them is less than exciting; and, as you have seen above, they have the same format in every class. That means software could be written to generate them automatically.

You instructor did that one afternoon. I wrote a `ClassMaker` class that inputs the shell of a class, with just the name and a list of variables, and produces the constructors, accessors, and `toString()` from a single button press. It is publicly available for your benefit, but you are only allowed to use it after you can write those methods correctly from memory. The reason is that the mechanics of accessors, constructors and `toString()` are those of the majority of all methods and until you understand how to write them, everything else will be hopeless. Hopeless. Get it? Don’t use the ClassMaker until you can write these methods yourself without peeking. The easy way, is not easy in the long run.

**ClassMaker Input and Output**

Once you are able to write constructors, accessors, and `toString()` from memory (usually everyone can do it by the fourth or fifth week of class; it just takes practice), you are ready to use the `ClassMaker`. It takes some of the drudgery out of creating a new class, and any repetitive task that can be automated should be.

You can find the ClassMaker at: `http://www.willamette.edu/~levenick/classMaker/makeClass.html`

Listing 5.6 is the input to the ClassMaker to produce the `Account` class in Listing 5.7.

Listing 5.6 Input to the ClassMaker for the Account Class

```
1    class Account {
2        int balance;
```
Listing 5.7 Output from the ClassMaker for the Account Class

```java
public class Account {
    protected int balance;
    protected String name;

    public Account() {} //empty default constructor

    public Account(int balance, String name) { //init'izing constructor
        this();
        this.balance = balance;
        this.name = name;
    }

    public int getBalance() {return balance;}
    public String getName() {return name;}

    public void setBalance(int balance) { this.balance = balance;}
    public void setName(String name) { this.name = name;}

    public String toString() {
        String returnMe = "I am a Account: ";
        returnMe += "\tbalance=" + getBalance();
        returnMe += "\tname=" + getName();
        return returnMe;
    } // toString()
}  // Account
```

Note that it produces the default constructor, accessors for both variables, and a toString() method that displays the values of both variables by using their constructors. You can copy and paste from the Web page into the Netbeans editor window. Read each line of code in it Listing 5.7. Become familiar with it; that means be aware of: 1) the name of the class, 2) each variable (its name and type), and 3) each method (its heading and body). Do that now. Notice anything strange? The setters are different than those earlier in the text. They do exactly the same thing as the previous setters in a different way; the details are in “Special to Java -- What is this?” below.

As an exercise, recreate the ATMApplet from scratch using the Classmaker to create the frameworks for Account, and Bank. Notice how much of the code is written for you. Do the same for the EyesApplet. Expect this to take several hours. Sorry. It’s good to practice; when I’m trying to learn a new programming environment or language, I will repeat very simple tasks over and over until I can do them effortlessly without running into problems/perplexities each time -- then I know I have mastered the process.

You should expect to discover that it’s much easier to do things the second time, plus this time what you’re doing will make more sense. Once the mechanics of using Netbeans becomes more
or less automatic you will have more cognitive capacity for the problems that will inevitably arise while programming.

**Constructors**

A peculiarity of OOP is that often much of the functionality of a class is accomplished by the constructors (or constructor chains). This statement will make more sense once you have had some experience with constructors.

Syntactically, a constructor declaration is like any other method declaration, with two differences. First, there is no return type (and no value can be returned). Second, its name must be the same as the class it is in.

Executing a constructor is just like executing any other method, but it happens automatically when you create an object of that type with matching parameters.

**Default Constructors**

A default constructor has no parameters. When the following line is executed:

```
Account myAccount = new Account();
```

these four things happen.

1. The new `Account` object is created, i.e. space is allocated for all its variables.
2. Those variables are initialized (either automatically to 0, or to values provided after =s).
3. The default constructor is executed.
4. The newly constructed `Account` is returned and stored in the `myAccount` variable.

There are thus two ways to automatically initialize the value of an instance variable; either use an assignment with the declaration `int balance=1000000;`, or insert an assignment statement in the default constructor:

```java
public Account() {
    balance = 17;
}
```

*Assuming you did both, what would the initial value of balance be? Hint: think about the order of the 4 steps above.*

**Account Class Including a Constructor With Parameters**

Usually, when you create objects, you wish to give them particular values. With only a default constructor, you are forced to first create the object and then set its values with accessors as in Listing 5.8.

Listing 5.8 Initializing with Accessors

```
1    Account myAccount = new Account();
2    myAccount.setName("Frodo");
```
These three lines can be compressed into one if there is a constructor that is passed initial values for the instance variables, as shown in Listing 5.9.

Listing 5.9 Initializing with a Constructor

```java
Account myAccount = new Account("Frodo", 1000000000);
Circle myCircle = new Circle(200, 100, 77);
```

For a Circle, one line replaces four. Obviously, it is fewer keystrokes and clearer to use the initializing constructor, and the ClassMaker writes it for you!

**Eye/FilledCircle/Circle Classes Including a Constructor With Parameters -- And Simplifications Appertaining Thereunto**

If we rewrite the Eye Applet using initializing constructors, there are a number of savings. Most obvious is where we create and initialize the Eyes as shown in Listing 5.10.

Listing 5.10 Initializing an Eye with Accessors

```java
Eye rightEye = new Eye();
rightEye.setX(600);
rightEye.setY(100);
rightEye.setRadius(100);
vs...
Eye rightEye = new Eye(600, 100, 100);
```

After using the default constructor, the position and size must then be initialized; three lines of code in initComponents(). With the initializing constructor those three lines disappear; see Listing 5.10. Better, by adding the color to the initializing constructor the color can be initialized when the Eye is created as well; Listing 5.11.

Listing 5.11 Initializing an Eye with A Constructor

```java
Eye rightEye = new Eye(600, 100, 100, new Color(200, 177, 200));
```

In this example, the initializing constructors for both Eye and the built-in class Color are used. The fourth parameter to new Eye() is the constructor for Color. You will recall that the parameter linkage mechanism first evaluates each actual parameter and then copies each value to the corresponding formal parameter. To evaluate the Color constructor, a new Color object is instantiated and then passed to the Eye constructor (which passes it on to the FilledCircle object that is created in it, see line 12 in Listing 5.12).

Listing 5.12 The Eye Class, Improved

```java
import java.awt.*;
```
This is an example of what this section mentioned at the start, namely the phenomenon that in
object programming, much of the work can be migrated to the constructors.

**Special to Java -- What Is this?**

**this**
The reserved word **this** has a special meaning in the context of an instance method, it is the
object which was sent the message that caused this method to be executed; or, shorter, the current
object. In the context of a constructor, **this** is the object that is being constructed.

The code written by the **ClassMaker** uses **this** to access instance variables. Compare the two
setters in Listing 5.13 which are copied from Listing 5.4 and Listing 5.7.

**Listing 5.13 Setters -- Two Techniques to Avoid Shadowing**

```java
public void setBalance(int nuBalance) {
    balance = nuBalance;
}
```
They both do exactly the same thing, namely assign the value of their parameter to the instance variable named `balance`. In the first version, the name of the parameter is `nuBalance`, so line 2 assigns the value of that parameter to the instance variable `balance` as desired. In the second version, the parameter is named `balance`, just like the instance variable. Thus, in the body of the second `setBalance()`, there are two different variables, both with the name “balance”. If the programmer, without thinking, attempted to set the instance variable named `balance` to the value of the parameter named `balance` by typing `balance = balance;` on line 6, it could cause a hard to find bug. When there are two variables with the same name defined in the same place, Java uses the one that is defined the closest (actually the one defined in the nearest enclosing scope, see “Variables II (varieties and scope)” below). In this case the parameter `balance` is defined closer (looking up the code from line 6), so it is used both times (it is said to `shadow` the instance variable). So, the value of the parameter `balance` is retrieved and stored back in the parameter `balance`, leaving the instance variable `balance` unchanged. To specify the instance variable `balance`, use `this.balance`.

The code the `ClassMaker` made for `Circle` included an initializing constructor, shown in Listing 5.14.

**Listing 5.14 The Initializing Constructor Written by the ClassMaker**

```java
public class Circle {
    protected int x;
    protected int y;
    protected int radius;

    public Circle(){}   //empty default constructor
    public Circle(int x, int y, int radius) {//initializing constructor
        this();   // invoke the default constructor
        this.x = x;
        this.y = y;
        this.radius = radius;
    }
}
```

Note the use of `this` in the initializing constructor.

**Line 9:** Invokes the default constructor; `this()` is the default constructor

**Lines 10-12:** Sets the three variables. `this.x` is the instance variable `x`

Because the parameters `x`, `y`, and `radius`, are the same as the instance variables, it uses `this` to store the values in the instance variables.
The \texttt{this()} Method

If there are initialization tasks that are being performed for every new instance of an object, they should be done in the default constructor. That way, later, when you (or other people) add additional initializing constructors, as long as they invoke the default constructor, the functionality of all the constructors can be preserved even when someone subsequently alters the default constructor. To invoke the default constructor, in another constructor, you say, \texttt{this()}. But it must be the first line of the constructor body; otherwise it will not compile.

Wait! What was all that in the previous paragraph? It seemed to be about avoiding possible future problems if someone added initializing constructors and then after that someone else changed the default constructor. If all you’re doing is trying to learn to program and writing tiny little programs that are just going to be discarded, \textit{who cares}?! Okay, right, of course not. Like much of Java, this only makes any appreciable difference when you are doing something big and complicated. This is what makes learning Java a bit difficult at first. If you have the feeling it’s more complicated than it needs to be to accomplish simple tasks, you are exactly right.

It’s a bit like if you want to build a little, simple bird feeder and your friend who is a machinist says, “I’ve got just what you need.”, and opens the door to a machine shop as big as a basketball arena, packed full with numeric-control machines, whirring and spinning ominously. And all you really need is a hand saw and a hammer. Overkill. On the other hand, if you were one day planning to build something complicated, like perhaps a Mars rover, or a better cell phone, or... you name it; then you might want to learn a bit about the more complex techniques.

Don’t worry about those details right now, just be aware of them, so if sometime you run into \texttt{this()} you won’t be completely flummoxed. By the way, the three programmers, the first who wrote the class originally, the second who modified the initializing constructor, and the third who subsequently modified the default constructor, might all be the same person at different times. They might all be you.

Details II

With those examples in mind, we are ready to confront more details.

Types

In Java, a type is either, a primitive type, a built-in class or a user-defined class. Some types you have worked with include \texttt{int, String, Applet, Account, Circle} and \texttt{Eye}. The first of those is a primitive, the next two are built-in and the last three user-defined. There are several other primitive types, many built-in classes, and potentially infinitely many user-defined classes.

Primitive Types

The primitive types are either numeric or non-numeric. There are only two non-numeric primitive types: \texttt{char} and \texttt{boolean}; these represent character and logical values, respectively. The numeric types are either, whole numbers, roughly like the integers (\texttt{long, int, short, byte}); or
decimals, roughly like real numbers (double, float). But there are infinitely many integers and uncountably infinite reals, whereas every primitive Java type is represented in a limited amount of space and so can only take on a finite number of values. The primitive types, along with their possible values and operators appear in Table 5.2.

<table>
<thead>
<tr>
<th>Type</th>
<th>Range of Values</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>long</td>
<td>(-2^{63} \leq x \leq 2^{63}-1)</td>
<td>+, -, *, /, %</td>
</tr>
<tr>
<td>int</td>
<td>(-2^{31} \leq x \leq 2^{31}-1)</td>
<td></td>
</tr>
<tr>
<td>short</td>
<td>(-2^{15} \leq x \leq 2^{15}-1)</td>
<td></td>
</tr>
<tr>
<td>byte</td>
<td>(-2^7 \leq x \leq 2^7-1)</td>
<td></td>
</tr>
<tr>
<td>double</td>
<td>(-1.8 \times 10^{308} &lt; x &lt; 1.8 \times 10^{308})</td>
<td>+, -, *, /</td>
</tr>
<tr>
<td>float</td>
<td>(-1.4 \times 10^{38} &lt; x &lt; 1.4 \times 10^{38})</td>
<td>+, -, *, /</td>
</tr>
<tr>
<td>boolean</td>
<td>{false, true}</td>
<td>!, &amp;&amp;,</td>
</tr>
<tr>
<td>char</td>
<td>any keyboard character</td>
<td>none</td>
</tr>
</tbody>
</table>

Table 5.2 Types, Values, Operators

For most applications ints will work fine for whole numbers and doubles for fractional numbers. There is no reason to use float, short, or byte, unless you discover you are out of memory; and that never happens (“What never? No, never! What never? Well... hardly ever!”). If you tried to count all the people on the planet, or keep track of the national debt, ints are too small, as \(2^{31}\) is only a little more than 2 billion. Fortunately longs would work just fine for those. For unlimitedly large numbers there is the BigNumber class.

**Numeric Types, Representation: Bits, Bytes and Powers of Two**

Some people have the idea that computing is all about bits and bytes, zeros and ones; and it is, underneath (just as life is all about chemistry, molecular machinery, underneath). Modern computing deals very little with bits and bytes, but there are times when you need to understand them. One of those times is if you want to understand how numbers are represented in Java and why they act the way they do.

The range of values for the type int is shown in Table 5.2 as \(-2^{31} \leq x \leq 2^{31}-1\)-- this has several implications and an informative cause. First, it means that if you need to store numbers larger or smaller than that, you must use another type, in this case, long. Second, if you add one to \(2^{31}\), instead of getting \(2^{31}\) as you might expect, you get negative \(2^{31}\) instead! Try it for yourself. Set a variable to 2 billion (2000000000) and then add it to itself and print the result. How to do this?

Type these lines into a main() method, or if your Applet is still on the screen, into init().

```java
int big = 2000000000;
int bigger = big + big;
System.out.println("2 billion + 2 billion=" + bigger);
```

On my machine this code it prints:

```
2 billion + 2 billion=-294967296
```
Which is certainly not 4 billion! Why does this happen? The explanation stems from the representation of ints.

An int is represented in Java as four bytes. A byte is 8 bits. A bit is the smallest possible unit of information; it has only two values, 0 or 1. So, a bit is just enough information to distinguish “yes” from “no”. Four bytes have 8+8+8+8 bits, that’s 32, so an int can take on $2^{32}$ different values (if you don’t know why, reread “Problem Solving Principle #1 -- Build a Prototype” in Chapter 1). Half those values are used for positive numbers (zero on up), the other half for negative numbers; half of $2^{32}$ is $2^{31}$. Since there is no negative zero (i.e. the least negative number is -1) the smallest number is - $2^{31}$, whereas zero is the smallest non-negative number, so that only leaves $2^{31}$-1 other positive numbers.

The types, long, short and byte, have eight, two and one byte, respectively (that’s 64, 16 and 8 bits); if you look at Table 5.2, their ranges correspond perfectly to that.

**Arithmetic Operators**

Table 5.2 shows four operators for doubles, the ordinary arithmetic operators, +, -, *, and / . They work just the way you would expect; when applied to two double operands, they yield a double value. Int operators when applied to two ints yield an int value. There are two int division operators, / and %. Int division is the division you might have learned in third grade, where $7/3=2$ with a remainder of 1. The / operator, applied to two ints yields the number of times the second goes into the first evenly, $7/3=2$. The % operator yields the remainder, $7\%3=1$.

**Mixed Expressions**

When the two operands of an arithmetic operator are an int and a double, the int is converted to a double and double operator is used. As you would guess, a double value results. Thus, $2*2$ evaluates to int 4, but $2*2.0$ evaluates to double 4.0.

The reason the int is automatically converted to a double and not vice versa is that the range of doubles is so much greater than that of ints. There are many doubles which simply cannot be represented as ints, but every int can be represented exactly as a double.

**Expressions**

All information in a Java program has a type. Information exists in both variables and expressions. Like a variable, an expression has a type and a value; unlike a variable, it does not have a name. Expressions are not declared and sometimes it is not obvious what their type is.

In the BNF thus far, expressions have only appeared in two places: on the right of the assignment operator and as actual parameters. That means that anything appearing either on the right of an assignment operator or as an actual parameter must be an expression syntactically.

Here are some int expressions:

- 17
- 1 + 1
1 + 2 * 3
1 + 2 - 3 / 4 * 5
(1 + 2 -3/4) * 5

The values are: 17, 2, 7, 3, and 15. The first three should be obvious. The value of 17 is 17, 1+1=2, and everyone knows that you do multiplication before addition. The value of the fourth hinges on how integer division works. This is because the / operator does integer division, discarding any remainder. This can be the cause of very subtle bugs, as will be seen below. So 3/4 = 0, 0*5=0, 1+2=3, and 3-0=0. The fifth expression uses parentheses to cause the multiplication to happen last, so (3+0)*5=15.

Precedence of Operators
In an expression with multiple operators, the question arises, “Which operator is applied first?”. In other words, which operation precedes which others? You are already familiar with precedence, since you know that 1 + 2 * 3 is 7. In ordinary arithmetic multiplication precedes addition. In computing, one says, “* has higher precedence than +”; it means just what you expect, without parentheses, multiplication happens before addition. Precedences appear in Table 5.3.

```
  .   -- the message dot
  (cast)
  !, unary –
  *, /, %, &&
  +, -, ||
  <, >, <=, =>, ==, !=
```

Table 5.3 Precedence of Operators - Higher Precedence Operators Are Higher in the Table
The rule is, if you’re worried that precedence is a problem, use parentheses.

BNF For Expressions
You may have noticed that the BNF for expression was missing... or not, whatever… here it is.

BNF 5.12 Expression

```
<expression> ::= <constant>
    | <variable>
    | <message expression>
    | <expression> <binary operator> <expression>
    | (<expression> )
    | <unary operator> <expression>
```

A recursive BNF production.

Notice that this is a recursive definition (if you’ve forgotten the term recursive, see “What classes will we need? What will they do?”, near the beginning of Chapter 3); it can generate
arbitrarily complex expressions. Binary and unary operators take two and one expressions as operands, respectively; they are defined as in BNF 5.13 and BNF 5.14.

**BNF 5.13 Binary Operators**

| <binary operator> ::= + | - | * | / | % | && | || | < | > | <= | >= | == | != |
|-------------------|---------------------|---------------------|---------------------|---------------------|---------------------|
| && is and || is or; these are boolean operators |
| <, >, <=, >=, ==, != are relational operators, they compare their operands and yield a boolean expression. The operator == is equals, != is not equals |

**BNF 5.14 Unary Operators**

<table>
<thead>
<tr>
<th>&lt;unary operator&gt; ::= -</th>
<th>!</th>
</tr>
</thead>
<tbody>
<tr>
<td>- is minus, as in -17, ! is not, as in !(x&gt;100)</td>
<td></td>
</tr>
</tbody>
</table>

Notice that expressions may be arbitrarily complex.

**Expressions Compatible With a Type**

In an assignment statement, the expression on the right of the assignment operator must be compatible with the variable on the left. The same is true of actual and corresponding formal parameters.

The simplest form of compatibility is identity. I.e. an int expression is compatible with an int variable -- you can always assign the value of an expression to a variable of the same type.

For now, all you need to know is that expressions of type int are compatible with double variables, so it is legal to assign an int value to a double variable (or to use an int expression as an actual parameter corresponding to a double formal parameter), but the reverse is not true (See “Mixed Expressions” above. Expressions of a type completely unrelated to a variable’s type can never be compatible; you can never assign a char or a String to an int, or an int or a double to a String. But, sometimes you can convert them by hand.

**Converting One Type To Another**

There are a number of different techniques to convert one type to and another when they are incompatible.

**String to int**

When input comes from a TextField it is always a String (the signature of getText() is public String getText()). The method that converts a String to an int is Integer.parseInt(String). It was illustrated in the section “Building and testing the prototype GUI”, in Chapter 3. There is a similar method for doubles.

**Object to String**

Any object can be converted to a String using toString() -- but, you already knew that!
An int (or any primitive type) can be converted to a String by concatenating it to the empty String, ".". Like this ""+17 becomes "17".

Casting
It is possible, in some situations, to force an expression to be a particular type. When you cast an expression, you are essentially saying to the compiler, "I know more than you; do what I say here!". The next section illustrates casting.

An Example -- Random Circles at Random Locations
Let’s say you decide to create a bunch of Circles at random locations and sizes and display them. There is a random number generator that you can access by saying Math.random() -- it returns a random number in the interval [0,1) whose type is double. To place a Circle at random requires random x and y values. To make it a random size requires a random radius value. Assume you want Circles with centers in a square 500 pixels on a side with radii up to 200 pixels; then you would need two random ints between 0 and 499, and one between 1 and 200. Since random() provides doubles in [0,1) you need to map from that small interval to the larger ones. The best way to do that is with a method (so as not to have to write the conversion code over and over -- and if it turns out to have a bug, you will only have to fix it one place... plus, it might be useful later).

It is easy to perform this mapping, simply multiply. 500*0 = 0 and any number < 1 times 500 is less than 500. So the method might look like Listing 5.15.

Listing 5.15 A Method to Provide Random ints with Bugs
1    int rand(int max) {
2        return Math.random() * max;
3    }

This method will not compile because the expression has type double, but the method heading declares the return type as int.

The problem is, that when you multiply a double by an int you get a double. Casting a double as an int simply truncates anything after the decimal point, so the obvious solution is to cast that expression as an int, see Listing 5.16.

Listing 5.16 A Method to Provide Random ints with a Precedence Error
1    int rand(int max) {
2        return (int) Math.random() * max;
3    }

This has a precedence error; it compiles, but always returns zero. The way to understand why is to realize that *, (int), and the dot after Math, are all operators and will be applied left to right in order of precedence. So, first the random() message will be sent, returning a value in [0,1),
then the cast, (int) will be applied, converting the value to an int 0 (through truncation), finally the multiplication will result in 0.

As always, the way to fix precedence problems is with parentheses, as shown in Listing 5.17. The parentheses cause the multiplication to happen before the cast, so everything is copasetic.

Listing 5.17 A Method to Provide Random ints that Works!

```java
1    int rand(int max) {
2        return (int) (Math.random() * max);
3    }
```

Insert this method into your revised EyeApplet class, and replace the body of `paint()` with the line:

```java
1    (new Circle(rand(500), rand(500), rand(200)).paint(g);
```

This will draw a different Circle each time you resize, or drag the window, or push a button in the Applet. Try it. When it works, make 5 copies of that line as the body of `paint()` and execute that.

**Random FilledCircles**

If you feel like playing a bit more before continuing on with this endless progression of detail, try this. Create and display FilledCircles instead of Circles. The FilledCircle constructor requires a fourth parameter, of type `Color`. To generate a random `Color`, simply give it three random values in the range [0,255] as parameters, as shown in Listing 5.18. Add this method to your EyeApplet.

Listing 5.18 randColor() Returns a Random Color

```java
1    Color randColor() {
2        return new Color(rand(256), rand(256), rand(256));
3    }
```

Then, modify `paint()` so it contains:

```java
(nw FilledCircle(rand(500), rand(500), rand(50), randColor())).paint(g);
```

as many times as you want random FilledCircles displayed.

If you wanted 20 FilledCircles, you could copy that line 20 times; or, you could write the `for` loop in Listing 5.19.

Listing 5.19 A Loop to Create and Display 20 Random FilledCircles

```java
1    for (int i=0; i<20; i++)
2        (new FilledCircle(rand(500), rand(500), rand(50), randColor())).paint(g);
```
The details of this loop will be explained in Chapter 8, but for now just think of it as “abracadabra”. Try it with 100FilledCircles (yes, change the 20 to 100). Or 100. Does it slow down much?

Variables II (Varieties and Scope)
There are five different varieties of variables. These include instance variables, class variables, parameters, method variables and loop variables. These will be addressed in order of how often they have appeared in this text so far. They differ in where they are stored, how they are initialized, how long they exist, and where they are visible. This last is referred to as their scope.

Instance Variables
By far the most common variables you have seen so far, and will encounter in object programming are instance variables. See “Variables I (state)” above for details. They are declared outside of any method and are visible everywhere in the class. They may use = <expression>, to initialize them, but if not, they are initialed to zero when the instance is created (before the constructor is executed).

Local Variables: Parameters, Method Variables, and For Loop Variables
There are three kinds of local variables. Unlike instance variables, their scope is the method or loop in which they are declared, and only exist while it is being executed.

Formal parameters have the form of variable declarations, namely <type> <identifier>. They are visible in the method they exist in. Their values are provided as part of parameter linkage when the method is invoked, and cannot generally be determined at compile-time. There are many examples in the text so far and details are in “Parameters (actual, formal, linkage)” above.

Method variables are declared within the body of a method. They have appeared in Listing 3.12 (Bank theBank), Listing 5.4 (Account myAccount), and Listing 5.5 (String returnMe). They exist only in the body of the method and must be initialized.

The only for loop variable was in Listing 5.19, (int i). Loop variables only exist within the loop they are declared in, and must be initialized when declared.

Class Variables
So far, no class variables have been used. They are not used very much in elementary programming; some people program for years and never use them. They are useful in certain situations though and you might run into one somewhere. Syntactically, class variables are exactly like instance variables, except they have the keyword static in front. Class variables do not belong to any instance, but instead to the class -- hence the name. They are used when there is information that must be accessible from every instance, but which does not belong to the instances. A class variable exists as long as the program is running and is visible from every instance. Variables are summarized in Table 5.4.
Table 5.4 Variable – Types, Scopes and Initialization

<table>
<thead>
<tr>
<th>Variable Type</th>
<th>Scope</th>
<th>Initialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>instance variable</td>
<td>entire class</td>
<td>auto, to 0 or by assignment by parameter linkage</td>
</tr>
<tr>
<td>formal parameter</td>
<td>body of method</td>
<td>must assign initial value</td>
</tr>
<tr>
<td>method variable</td>
<td>body of method</td>
<td>must assign initial value</td>
</tr>
<tr>
<td>loop variable</td>
<td>body of loop</td>
<td></td>
</tr>
<tr>
<td>class variable</td>
<td>all instances</td>
<td>like instance</td>
</tr>
</tbody>
</table>

Example -- Serial Numbers

Some people, when they first start object programming feel somewhat uncomfortable about having many nearly identical objects of the same class, and would like to be able to keep track of which is which. One easy way to distinguish between nearly identical objects in the world, is to attach a serial number to each one. The first one is number 1, the second, number 2, and etc. The same may be done with Java objects. Here’s how.

Each object must keep track of its serial number, so there must be an instance variable; e.g.

```java
int serialNumber; -- this will be set to 1 for the first object instantiated, 2 for the second, etc.
```

There must also be a class variable to keep track of the next serial number to be assigned; like this:

```java
static int nextSerialNumber=1;
```

Then, in the default constructor, there must be one line added.

```java
serialNumber = nextSerialNumber++;
```

This is shorthand for

```java
serialNumber = nextSerialNumber;
nextSerialNumber = nextSerialNumber + 1;
```

The first line assigns the current value of the class variable `nextSerialNumber` (1 the first time) to the instance variable `serialNumber`. The second line increments the class variable `nextSerialNumber`.

That’s all it takes. To see that it works add this line as the second in `toString()`:

```java
returnMe += " my serial number is: " + serialNumber;
```

Modify the `Circle` or `Account` class (use the one with the `main()` method to make your task simple), and test to see that this works.

If you tried to use an instance variable to keep track of the serial number, you would discover you could not make the numbering work correctly. Consider this code:

```java
public class BrokeSerialExample {
    int nextSerialNumber=1;
    int serialNumber;

    public BrokeSerialExample() {
        serialNumber = nextSerialNumber++;
    }
```
Here the `static` has been deleted, so `nextSerialNumber` is an instance variable instead of a class variable. Thus, every instance will have a copy of the `nextSerialNumber` variable and each will be initialized to 1 before being incremented, so every instance will have a serial number with a value of 1.

**Conventions**

There are a number of conventions which are not required by the compiler, but that make it much easier to program. These are entirely arbitrary, but are pretty much standard in the industry.

**Naming Conventions**

Identifiers should convey information; they should tell what they are or what they do. This includes class names, method names and variable names. Typically, variables are nouns, methods are verbs. Accessor names start with `get` or `set` and then the variable being accessed, e.g. for `balance` we have `getBalance()` and `setBalance()`.

**Case Conventions**

Class names begin with upper-case letters. Instance names begin with lower-case letters. The second and any subsequent English word in an identifier starts with an upper-case letter. Constants are all upper-case.

**The Importance of Good Names**

Descriptive names can make the difference between being able to debug your program and not. The reason is simple, cognitive capacity. People can only keep in mind around 5 things at once. If you name your variables that mean balance and radius, Frank and Ernestine, then you are squandering 2 of your precious 5 on remembering which means which. When the program is extremely simple, this is not a big problem, but if it is just at the limit of the programmer’s capacity, this could lead to disaster.

**Recapitulation**

As mentioned in the first chapter there are a few dozen concepts that must be understood, at least vaguely, before a person can program in Java. Most of them have been covered in this chapter (if you’re feeling a bit overwhelmed, please be patient, it gets easier). Here’s a rough list; this would be suitable to read every night after programming until it is all very obvious. It appears, first as a list, then with explanations. When you can remember the explanations by looking at the list, then you can stop looking at it.

**Information**

- types
- values
- variables
- expressions
Language Elements

- classes
  - variables
  - methods
  - constructors

- objects

- statements

- identifiers

- methods/messages
  - signatures
  - parameters (formal/actual)
  - parameter linkages

- syntax

- semantics

Process

- editing
- compilation
- execution
- debugging
- prototyping

Here’s that same list with some explanatory text.

Information

Types
All information in a Java program has a type. There are primitive types, built-in classes and user-defined classes. It is possible to change types by casting.

Values
Expressions have values. To compute the value of an expression it is evaluated. Every type has a range of legal values.

Variables
Variables hold information. Every variable has a name, a type and a value. A variable only holds one value at a time. There are five different kinds of variables: instance, parameter, method, loop, and class.

Expressions
Variables and constants may be combined in arbitrarily complex fashion to form an expression. Syntactically, expressions appear to the right of assignment operators and as actual parameters (Thus far. Later on you will see them other places.).
**Language Elements**

**Classes**
A class is a template for objects of that type. It includes both variables and methods. Every object of that type has all the instance variables and can use all the methods declared in it.

If a method is declared `static`, it is not an instance method, but a class method. If a method does not use any state information from an instance (i.e. any instance variables) it can be made into a class method.

**Variables**
Every instance of a class has its own copy of each instance variable. All instances share access to class variables which are stored in the class itself (interestingly, classes are also objects, they are instances of the class `Class`).

**Methods**
Methods have a heading and a body. The heading specifies the type, name and parameters of the method. The method body is a single block statement, which is a pair of `{}`s around a series of statements. To execute the method, Java executes each of those statements in order.

**Constructors**
Constructors are typeless methods with the same name as the class that are executed when a new object of that class is constructed.

**Objects**
Objects (also called instances) of a particular class are created by saying

```java
new ClassName();
```

This is referred to as instantiation. When an object is instantiated, the constructor corresponding to the signature of the new message is executed.

**Statements**
Statements are what accomplish most of the action when a program executes. So far these statements have been presented: assignment statement, return statement, block statement, and message statement.

**Identifiers**
Identifiers are Java names. They start with a letter, and are composed only of letters, digits and underscores.

**Methods/Messages**
When you send a message to an object, it invokes the method in that class with the same signature. First it performs the parameter linkage, then executes the method body.
Signatures
The signature of a method is its access type, return type, name and parameter types.

Parameters (formal/actual)
The parameter in the method declaration is the formal parameter, the one in the message is the actual parameter.

Parameter Linkages
Each actual parameter is evaluated and its value copied to the corresponding formal parameter.

Syntax
The syntax of Java is defined by a set of BNF productions. Any source code not matching this grammar symbol for symbol is deemed to have compiler errors.

Semantics
The semantics of a statement is the action it performs when it is executed. An experienced programmer has internalized the semantics of enough of the constructs of the language that solving routine problems is easy.

Process

Editing
Editing is when the programmer is inputting or changing source code (classes).

Compilation
Compilation is when the compiler is checking the syntax of a class. Errors at this stage are compile-time errors and are either lexical or grammatical. Lexical errors happen when the compiler does not know what an identifier means; the most common causes are forgetting to declare variables, or typos. Grammatical errors occur when the syntax of the source code does not match the BNF description of the language precisely.

After verifying the syntax of a class, the compiler converts the source code to byte code. Assuming the source code is in a file called Foo.java, the byte code will be put in a file called Foo.class in that same directory.

Execution
To execute a program, the byte code is interpreted by the Java Virtual Machine. This is when the work of the programmer comes to fruition. The semantics of the various methods are carried out to achieve some desired result. Errors here are run-time errors, and appear as Exceptions.

Debugging
Debugging is the process of removing errors from a program. It is the most time consuming and frustrating aspect of programming. Any nontrivial program has multiple errors. Only novice
programmer imagine that one can program without bugs. Like dropping the balls when juggling; it happens. One important skill in programming is learning to write code that is easy to debug.

**Prototyping**

Building simple prototypes and adding functionality as the previous prototype works is perhaps the most important way to make debugging simple; there are simply less places to look for the bugs.

**Conclusion**

Programming in Java is accomplished by writing classes. Classes define the information objects of that type can store (by declaring variables) and the actions those objects can carry out (by declaring methods). Methods have a heading and a body; the former defines the signature of the method, the latter defines its action.

Although there are many programming constructs and statements, the only three that alter the state of the computation, the only three that really do anything, are input, output, and assignment; all the others are organizational, organizing both the structure of the program and which of the big three are executed in what order and how many times.

The standard set of methods that all classes should have, including constructors, accessors and `toString()`, are written automatically by the `ClassMaker`. Any others you will have to write by hand.

This chapter presented many of the details of Java programming. If it succeeded, you, the reader, are beginning to understand the interplay of the two dozen odd concepts that make up most of programming. If not, my apologies; with luck, perhaps after a bit more practice this stuff will fall together for you.

**Review Questions**

5.1 Why are initializing constructors useful?
5.2 What is `this`?
5.3 What is `this()`?
5.4 Why are good names important?
5.5 What will this output?

```java
byte x=127;
x++; System.out.println("x= " + x);
```

5.6 Name 4 varieties of variables. What are their scopes?
5.7 What does it mean for one variable to shadow another?
5.8 What is the type and value of:

```java
17
3.141
0.1*30
```
5.9 What language does this BNF generate?

\[
\begin{align*}
\langle S \rangle & ::= \langle A \rangle \ \langle B \rangle \\
\langle A \rangle & ::= a \ | \ a \ \langle A \rangle \ \langle B \rangle \\
\langle B \rangle & ::= b 
\end{align*}
\]

5.10 What do the symbols: ::=, |, [], <> [x]* mean in BNF

**Programming Exercises**

5.11 You can convert an `int` to a `String` by pasting it onto `""`. Try out this by

```java
System.out.println("" + 17);
```

You should see 17 print. Now try

```java
System.out.println("" + 17 + 17);
```

What goes wrong? Hint: you can fix it with parentheses.