Chapter 1 BEHAVIOR — METHODS

Your classes are defined to the world in terms of their behavior. Even your state is declared in terms of behavior — you use public methods to allow clients to access your instance variables (unless you violate virtually every book on object-oriented [OO] and Java programming and make your instance variables public). You do this because the principle of encapsulation states that classes should hide their data. So while objects are partnerships between individual instance data and behavior, the behavior gets all the glory, while the data hides behind the scenes.

Behavior is a two-way street. Implementation of behavior is done with methods; invocation of behavior is made by calling those methods, also referred to as sending a message. This book prefers the term "message" in the pure sense that when you make a method call, you are sending a message to an object to ask it to perform some behavior. How messages are sent and how the corresponding behavior is implemented in methods are very closely related.

This chapter and the next define a series of behavioral patterns that will aid you in organizing your class appropriately. You should already have defined your public methods during your object design phase, so that your work is partly done. You could implement those public methods and

conceivably be done with the development of your class. It would work, but it probably would not be very maintainable.

Applying these behavioral patterns will result in the creation of additional private or protected methods. This will help your class better declare how it accomplishes its published goal. It will also make your class more flexible. You achieve this by breaking up the tasks representing the public methods into understandable pieces. You also use consistent patterns to help guide an educated reader of the class through its intent.

Methods

The history of software development started with writing single programs that executed from the top down, start to finish. This progressed to the ability to define subroutines, or functions, which were scoped either globally, to a source module level, or even to another function. Finally, with object-oriented languages, we have moved to class-scoped methods. In Java, *all* code is scoped to a class. There is no such thing as a global function like there is in C++. And to build a class, you must code methods.

It would be entirely possible to code a complete class within its constructor method or a complete application within a `main()` method. Java does not restrict you from being so obnoxious — the Java *language* allows you to code as you wish. But the Java *class library* demonstrates that classes must be broken into separate methods to be useful. If you are coding an applet, for example, you may need to code `init()`, `stop()`, `start()`, `paint()` and `destroy()` methods. Each of these methods provides an important chunk of functionality to the applet.

I too will do my best to get you to code lots of methods. There is a distinct advantage to breaking up code within your class in specific ways. With long methods, you increase code duplication, limit reuse, and make your class difficult to decipher and maintain.

You might be thinking, "Doesn't performance suffer if you have lots of short methods?" Indeed, invoking a method in Java is a moderately expensive operation. Ironically, though, if you have performance problems in your code, you cannot effectively measure the source of your performance problems with large methods. Small methods give you the granularity you need to be able to isolate performance bottlenecks. For an in-depth discussion of how performance relates to *Essential Java Style* patterns, refer to Appendix A.

There is also a point at which an excessive number of methods makes a class too cumbersome to manage. But the possibility also exists that if your class has too many methods, another class may need to be defined.

In summary, there is a happy medium between lots of very small methods and few very long methods. Coding methods is about organizing your class to achieve the goal of optimal flexibility and maintainability. This section provides patterns for proper method organization.

COMPOSED METHOD

Answers the Question	How do you divide a class into methods?
Solution	Create small methods, each of which accomplishes a single task that is concisely represented by the method name.
Category	Behavioral
Related Patterns	Method Comment Intention-Revealing Method Name

The COMPOSED METHOD pattern is the centerpiece of this book. If you absorb one thing from *Essential Java Style*, make it how to use COMPOSED METHOD to organize your classes. Good class organization is what effective maintenance is all about.

A simple mantra to remember for class development is, "make it run, make it right, make it fast." COMPOSED METHOD and the remainder of the patterns in this book are the "make it right" part of the mantra. Your development should follow all three steps as described under the next three subheadings.

Making It Run

Making it run (correctly) is your most important goal in developing any software. If the code is beautiful but does not run, it is utterly useless.

Your initial class development should take the following steps:

1. Determine and name your public methods. Classes are about providing behavior to client objects. If you don't understand how your class will expose its behavior to other classes, you have missed an important step in object design.
2. Define the attributes that class instances will store in terms of instance variables.
3. Code the public methods. Typically there will be few additional methods at this point, because the most rapid way to code a method is top down, through to completion.

At this point, you will have a class that can be used by client code — "make it run." Is that enough?

While your class may run just fine, you will not have a class that can be easily maintained or extended. The methods that you have coded in step 3 will be fairly long. Are these long methods inherently bad? Code in the method itself can answer this question. I take the view that code speaks volumes, and if you don't like what it's saying, there is a problem.

First, aesthetics of presentation are important. If I cannot easily read code, my ability to rapidly maintain it will be diminished. White space, consistent indentation, and breaking things up into legible chunks go a long way towards improving readability. If I have a several-hundred-line method that spans multiple screens or pages and seems to go on forever with many things going on like this overly long sentence, it will take the reader considerably longer to even scan the method to get the whole picture. Page up, page down, bookmark, page up, page down, return to bookmark — What's going on here?

If I had condensed all the paragraphs in this book into a single paragraph, eliminating all of the paragraph headers, I guarantee that you would not have purchased the book. As it is, you may regret having bought this book, but at least it is organized well enough so that you can find your way around. The book speaks its intent by its chapter and topic organization.

Having long methods also means that the methods are doing lots of different things. This leads to negative side effects such as the need for excessive comments (see METHOD COMMENT) or generic, bland method names (see INTENTION-REVEALING METHOD NAME).

More importantly, though, coding long methods violates good object-oriented design principles by lumping scads of tasks into a single method. Your capacity for reuse within the class is diminished — the more tasks within a method, the more specialized it will be. Specialization minimizes the potential for inheritance and reuse and can also increase the amount of code duplication.

Small methods, which limit the number of things going on, end up being vastly more readable and easily replaceable or extensible. Smaller methods are also tested more easily. To properly test a method, you must prove that it produces the expected results for all possible inputs and initial object states.[1] These expected results include not only the return value of the method, but also the resultant object state. This is an excruciatingly painful task, as you might expect, and most people have neither the time nor the patience for it. Nevertheless, the fewer things that are going on within a single method, the less there is to prove.

Making It Right

After you have completed the first three steps in the development cycle and you have a handful of lengthy methods, your job is to "make it right." Making it right means applying the patterns in this book to your class to make it as easily readable and maintainable as possible. COMPOSED METHOD is the starting point.

---

1 Arbib, M.A., Kfoury, A.J., Moll, R. N. *A Basis for Theoretical Computer Science.* New York, NY: Springer-Verlag, New York Inc., 1981, p. 104.

Listing 1.1 provides an example of a fairly short method that does too much.

Listing 1.1 *COMPOSED METHOD Example — Before*

```
public void createCustomerCSV()
   throws IOException
{
   String tempFilename = null;
   for (int i = 0; i < 1000; i++)
   {
      String filename = "temp." + i;
      File file = new File(filename);
      if (!file.exists())
      {
         tempFilename = filename;
         break;
      }
   }
   if (tempFilename == null)
      throw new IOException(
         "Could not create temp file");

   FileWriter output =
      new FileWriter(tempFilename);
   try
   {
      Iterator iterator = customers.iterator();
      while (iterator.hasNext())
      {
         Customer customer =
            (Customer)iterator.next();
         output.write(customer.getName()+",");
         output.write(customer.getCity()+",");
         output.write(customer.getCountry()+"\n");
      }
   }
   finally
   {
      output.close();
   }
}
```

This `createCustomerCSV()` method is doing three separate tasks. It derives a legitimate temporary file name for output, loops through a list of customers, and writes information from each customer to the output file in a CSV (comma-separated values) format. If I were to comment the chunks in this method properly, I would need three comments just to guide the reader along.

Using COMPOSED METHOD to reorganize, or "refactor," the method, you might end up with the three methods shown in **Listing 1.2**.

Listing 1.2 *COMPOSED METHOD Example — After Refactoring*

```
public void createCustomerCSV(String filename)
   throws IOException
{
   FileWriter output =
      new FileWriter(filename);
   try
   {
      Iterator iterator = customers.iterator();
      while (iterator.hasNext())
         writeCustomer(output,
                       (Customer)iterator.next());
   }
   finally
   {
      output.close();
   }
}

public String getTempFilename(String prefix)
   throws IOException
{
   for (int i = 0; i < 1000; i++)
   {
      String filename = prefix + "." + i;
      File file = new File(filename);
      if (!file.exists())
         return filename;
   }
   throw new IOException(
      "Could not create temp file");
}

private void writeCustomer(
   FileWriter output,
   Customer customer)
   throws IOException
{
   output.write(customer.getName()+",");
   output.write(customer.getCity()+",");
   output.write(customer.getCountry()+"\n");
}
```

The benefits of this reorganization are many. Client code will have the flexibility to create the CSV file with whatever filename is desired, or it can call `getTempFilename()` to provide the filename parameter to `createCustomerCSV()`. If the algorithm in `getTempFilename()` is deemed inadequate, it can be easily overridden with a better one in a subclass. Also, if more fields from Customer are required to be written to the CSV record, the maintenance programmer can easily locate the single method (`writeCustomer()`) that manipulates the customer data.

Finally, and most importantly, the code in each of the three methods in **Listing 1.2** can be immediately understood. Not only is the code brief enough to visually recognize, but conceptually the method name tells us what each method is accomplishing. Conversely, if you cannot look at a method and rapidly determine what it is doing, then it is doing too much.

After applying this pattern, the bulk of your Java code (excluding repetitious processing, such as doing user interface layouts and setup) will be divided into short, concise methods. Most of my methods are from 5 to 10 lines of code each.[2]

Making It Fast

It's to be hoped that you will not have to pay homage to this part of the mantra very often. But sooner or later, someone will come along and complain about the speed. It is Java, after all.

If you do encounter performance problems, the biggest mistake you can make is to assume you know where the problem is. I have incorrectly assumed that a specific method was the bottleneck. I spent lots of time fixing the presumed source of my performance problem, only to make my code uglier and to find out that my assumption was incorrect. As my high school math teacher Mr. Brune often repeated, "When you assume, you make an *ass* out of *u* and *me*."

Use a tool to monitor performance. If your code was "made right," performance tests will point out the precise source of the problem. With large multipurpose methods, you will have no way of determining just which piece of a method is causing the bottleneck.

Modifications to increase performance usually are made at the expense of easily understood code. You will need to add comments to explain why things were done that way. Your code will become less maintainable. The implication of these downsides is that you should always save performance modifications until the last step. Even then, only implement them as a last resort.

For more information on performance issues, refer to Appendix A, "Performance."

2 Surprise — pretty much all that we are doing here with Composed Method is classic functional decomposition. Not that there's anything wrong with that.

CONSTRUCTOR METHOD

Answers the Question	How do you represent instance creation?
Solution	Provide a constructor for each valid way to create an instance; do not provide constructors that allow creation of invalid objects.
Category	Behavioral
Related Patterns	None

All classes have the capability to produce new instances. If you do not provide any constructors, Java provides what is known as the default constructor.[3] The default constructor is a no-argument method with the same name as the class; it returns a new instance of the class.

However, if you explicitly code *any* constructors, that's all you get — a "no-arg" constructor is not available to you, and any attempts to refer to one will result in a compile-time error. This is nice to know if you want to prohibit the creation of objects with no parameters. Line [1] in the `main()` method of **Listing 1.3** tries to call a nonexistent no-arg constructor to create a new Customer object.

Listing 1.3 ***Missing No-Arg Constructor Example***

```
// BROKEN CODE!
public class Customer
{
   public static void main(String[] args)
   {
(1)        Customer customer = new Customer();
      customer.display();
   }

   String name = "<no name provided>";
   public Customer(String _name)
   {
      name = _name;
   }
   public void display()
   {
      System.out.println(name);
   }
}
```

3 The term "default constructor" is a bit overloaded, so I am compelled to provide the appropriate definition. You will find other sources that use the term "default constructor" to refer to a constructor with no arguments. This use of "default" is misleading — a default anything is the fallback in case something is not provided. JavaSoft calls arguments with no constructors simply "no-arg constructors," which isn't a great term but is at least accurate. In this book, a default constructor is the no-arg constructor that is provided if none is explicitly defined.

Compiling **Listing 1.3** will result in the following error, because the constructor `Customer(String)` was provided but `Customer()` was not:

```
Customer.java:5: No constructor matching Customer()
found in class Customer.
      Customer customer = new Customer();
                          ^
```

So what kind of constructors should you provide? Usually your design will lead to the need for only a few constructors in each of your classes, and often only a single constructor is needed. The question that you must answer is: "At what point is my class valid?" In most cases, the set of CONSTRUCTOR METHODS should be all the ways to produce valid classes.

You might choose to provide only a no-arg constructor, allowing the individual attributes of a new object to be populated via its setter methods. This technique is shown in **Listing 1.4**.

Listing 1.4 ***Setting Attributes After Instance Creation***

```
Manager manager = new Manager();
manager.setName("Blow, Joseph");
manager.setSSN("999-13-1349");
manager.setContractedRate(40);
```

As long as the resultant object of the no-arg constructor is in a valid state, this is fine. Normally, though, your object requires additional data to be set in order for the object to be in a valid state.

In **Listing 1.4**, what if the client neglects to set the contracted rate? Calculations against the manager that require the rate will either produce incorrect results or cause an exception. Somehow, you have to alert client developers as to which fields need to be set, and hope they follow your comments.

Instead of trusting client developers, provide a set of constructors that represent the ways to create only valid instances. Required data is passed into the object via constructor parameters. Two examples of how to create the Manager object are shown in **Listing 1.5**.

Listing 1.5 ***Constructors Creating Valid Instances***

```
public Manager(String name,
               String ssn,
               int contractedRate)
public Manager(String name,
               String ssn,
               int contractedRate,
               boolean hasGoldenParachute)
```

CONSTRUCTOR METHOD answers the question: "How do I create a valid instance of this class?" for client developers. If you follow javadoc conventions, their answer is the list of methods available in the Constructor Summary of the javadoc pages produced for your class.

CONSTRUCTOR PARAMETER METHOD

Answers the Question	How do you set instance variables from the parameters to a CONSTRUCTOR METHOD?
Solution	Use DIRECT VARIABLE ACCESS; create a private `set()` method if more than one constructor needs to set common parameters.
Category	Behavioral
Related Patterns	Indirect Variable Access Direct Variable Access Default Parameter Values

Many of your constructors will take parameters of some sort. But what is the best way to set the object state from these parameters?

If you are using INDIRECT VARIABLE ACCESS, you might be tempted to use your setter methods to store the parameter data from the constructor in the object, as in **Listing 1.6**.

Listing 1.6 *Setters Used for Constructor Initialization*

```
public Manager(String name,
               String ssn,
               int contractedRate)
{
   super(ssn, name);
(2)   setContractedRate(contractedRate);
}
```

In [2] of this example, the operation of setting the `contractedRate` attribute is delegated to the `setContractedRate()` setter method.

Occasionally, you will come across the special case where attributes need to be set differently on object instantiation than after the object has been created. An example is the distinction between an empty object being populated from a database versus an object being newly created by an application. In the first case, you simply want to use the data retrieved from the database to directly set the attributes. In the second case, you might create a partially complete object and then trigger a database update operation when attributes are set.

With this in mind, you should avoid using the setters within the constructor. Instead, use DIRECT VARIABLE ACCESS to set the attributes (**Listing 1.7**).

Listing 1.7 *DIRECT VARIABLE ACCESS in Constructor Initialization*

```
public Manager(String _name,
               String _ssn,
               int _contractedRate)
{
   ssn = _ssn;
   name = _name;
   contractedRate = _contractedRate;
}
```

If you have more than one constructor or more complex operations are required to set the attributes, provide a private CONSTRUCTOR PARAMETER METHOD. This method takes all of the parameters from the constructor and does whatever operations are necessary to directly set the parameters. Call this method `set()`. **Listing 1.8** demonstrates the use of CONSTRUCTOR PARAMETER METHOD for the Manager example.

Listing 1.8 *CONSTRUCTOR PARAMETER METHOD Example*

```
public Manager(String name,
               String ssn,
               int contractedRate)
{
   set(name, ssn, contractedRate);
}

private void set(String _name,
                 String _ssn,
                 int _contractedRate)
{
   name = _name;
   ssn = _ssn;
   contractedRate = _contractedRate;
   baseSalary = contractedRate * 2000;
}
```

If you need to provide constructors that use default values for required attributes, use the pattern DEFAULT PARAMETER VALUES instead.

DEFAULT PARAMETER VALUES

Answers the Question	How do you set parameters to default values?
Solution	Overload the method with all combinations of required parameters. Delegate from the more specific methods with fewer parameters to the methods with more parameters, ultimately delegating to the method that does the actual work.
Category	Behavioral
Related Patterns	Constructor Parameter Method

If you are a C++ developer, you are familiar with the ability to define default arguments:

```
int setCursorTo(int x=1, int y=1)
{
   // ...
}
```

This capability is a shortcut for supplying function overloading. In this example, `setCursorTo()` can be called in three different ways:

```
setCursorTo();     // both x and y default to 1
setCursorTo(5);    // x is set to 5; y defaults to 1
setCursorTo(5, 4); // x is set to 5; y is set to 4
```

Java does not provide this feature by design. A possible substitute solution in Java would be to test each parameter and initialize the parameters that are passed in as null. This is a poor solution, as it adds a lot of code and overhead to your methods. DEFAULT PARAMETER VALUES defines a better technique for accomplishing this: you provide multiple methods representing all possible initialization scenarios.

To accomplish this, first create a method that takes all possible parameters and does the actual work. Overload this worker method with methods, each of which takes one less parameter than the previous one, and delegates back to the method with the next-most parameters, supplying defaults as required. This sounds like a mouthful; a demonstration using the `setCursorTo()` example should clear things up:

```
int setCursorTo(int x, int y)
{
   // ...
}
int setCursorTo(int x)
{
   return setCursorTo(x, 1);
}
int setCursorTo()
{
   return setCursorTo(1, 1);
}
```

If you have any sort of conscience, you will use either the DEFAULT VALUE pattern or the DEFAULT VALUE METHOD pattern to help explain why x and y default to the value 1.

DEFAULT PARAMETER VALUES can also be used to provide default values, for instance, creation, as shown in **Listing 1.9**.

Listing 1.9 *DEFAULT PARAMETER VALUES Example*

```
public class Employee
{
   public Employee(String name)
   {
      this(name, 40);
   }
   public Employee(String name,
                   int hoursWorkedPerWeek)
   {
      // ...
   }
}
```

You will need to properly organize your method cascades. Determine which parameters are optional and which must be provided by client code.

First, code the method that takes all parameters, including defaults. Required parameters will appear to the left in the argument list, followed by the defaultable parameters. Then code the method with the next fewer default parameters; repeat until all methods have been provided. For javadoc presentation purposes, the method list is usually the reverse of how you will code things: start with the shortest method signature and move to the one with the most arguments.

SHORTCUT CONSTRUCTOR METHOD

Answers the Question	How can you simplify the construction of objects?
Solution	Provide a method that creates an instance of a new object, using its parameter as an initialization value.
Category	Behavioral
Related Patterns	Converter Method

This pattern is a bit more useful in C++ or Smalltalk, where you have the capability to do operator overloading. Operator overloading, a confusing feature at best in C++, is a powerful construct in Smalltalk that can make your code considerably more expressive. It can be abused, of course, as can just about any construct in any language, including Java.

In Java, without the benefit of operator overloading, there are not many good examples for SHORTCUT CONSTRUCTOR METHOD. The concept of the pattern is that you implement a shortcut method in one class that constructs and returns an object of another class. Information required to construct the new class may come either from the source class or from parameters passed to the SHORTCUT CONSTRUCTOR METHOD.

An example in VisualWorks Smalltalk is the shortcut creation of an Association object. An Association is simply an object that contains a key and value pair. A similar construct in JDK 1.2 is the Map.Entry interface obtained by enumerating a Map. In Smalltalk, you can create an Association object by sending it the message `->` with any other object as a parameter. The method in Smalltalk is:

```
-> anObject
   ^Association key: self value: anObject
```

Client code to create an Association looks like the following:

```
x -> y
```

In Java, consider a word processor application. Suppose you have instances of the class Word representing the individual words captured by a word processor. As words are typed, they are combined with other words to form a Sentence object.

Without SHORTCUT CONSTRUCTOR METHOD, to concatenate a word to an existing sentence and create a new sentence, you would code:

```
new Sentence(existingSentence, word);
```

Using SHORTCUT CONSTRUCTOR METHOD, you implement a method in Sentence that takes a Word as a parameter and returns a new Sentence object. **Listing 1.10** demonstrates how this is accomplished.

Listing 1.10 *SHORTCUT CONSTRUCTOR METHOD Example*

```
public Sentence cat(Word word)
{
   return new Sentence(this, word);
}
```

Concatenate is abbreviated as `cat`. It would be great if you could overload the + operator. Too bad Java won't let you. Using this SHORTCUT CONSTRUCTOR METHOD, your client code becomes:

```
existingSentence.cat(word);
```

This pattern is useful only if the operation is performed frequently in code, otherwise you are sacrificing understandability for little gain. Using this pattern makes it appear that your code is doing a little bit of magic, unless you are familiar with the idiom. Understanding that a method is creating a new object of a different class is not necessarily intuitive.

CONVERTER METHOD

Answers the Question	How do you represent simple conversion of one object to another with the same protocol but a different format?
Solution	Prefer the use of CONVERTER CONSTRUCTOR METHOD if possible. If not, create a method `asTargetClass()` and have it return a new instance of the target class.
Category	Behavioral
Related Patterns	Converter Constructor Method Shortcut Contstructor Method

If you are required to convert an object from a target class to a source class, you might be tempted to provide converter methods within the source class. For instance, to convert from a Grunt to a Manager (a promotion!), your solution might be the following method on Grunt:

```
public asManager()     //  AVOID THIS TECHNIQUE
{
   initialContractedRate =
      Math.round(getSalary() / 2000);
   return new Manager(name,
                      ssn,
                      initialContractedRate,
                      false);
}
```

But using this solution means that your source class, Grunt, is now dependent on the target class, Manager. If the constructors for Manager change, the Grunt class will have to change as well. If you add a new class, such as Contractor, you would now need to add an `asContractor()` method to Employee, and you would also want the same method on Manager. This increases coupling between classes from a code maintenance standpoint and tends to clutter your classes.

Often you will not have the ability to change the target class, in which case the use of CONVERTER METHOD cannot be avoided. If this is the case, provide a method name that follows the pattern `asTargetClass()` and have it return a new instance of the target class.

If you are able to modify the target class, the preferred solution is to provide a CONVERTER CONSTRUCTOR METHOD. If your conversion is used heavily within code, you may still opt to provide a CONVERTER METHOD on the source class as in **Listing 1.11**.

Listing 1.11 *CONVERTER METHOD Example*

```
public asManager()
{
   return new Manager(this);
}
```

The `asManager()` method in **Listing 1.11** should never need to change and is only used to slightly reduce the amount of code.

CONVERTER CONSTRUCTOR METHOD

Answers the Question	How do you represent the conversion of an object to another, possibly with a different protocol?
Solution	Provide constructor methods in the target class that take the source object as a parameter.
Category	Behavioral
Related Patterns	Converter Method

The ideal solution for converting one object to another is to simply overload the constructor of the target object. In other words, for each source to be converted from, provide a constructor in the form:

```
public TargetClass(SourceClass sourceParm)
```

Listing 1.12 provides an example that supports the promotion of a Grunt to a Manager.

Listing 1.12 *CONVERTER CONSTRUCTOR METHOD Example*

```
public Manager(Grunt grunt)
{
   name = grunt.getName();
   ssn = grunt.getSSN();
   contractedRate = grunt.getSalary() / 2000;
   hasGoldenParachute = false;
}
```

CONVERTER CONSTRUCTOR METHOD makes the target class dependent only on the technique for conversion. As long as the Grunt class provides the appropriate public methods, any changes to the conversion process are isolated to the constructor of Manager. The Manager class bears sole responsibility for understanding how a Grunt gets "promoted" to a Manager.

Provide a constructor in the target class for each source class to be converted from. Its sole parameter is the source class object.

A Note on Converting Strings and ints

Java provides several different ways to convert ints (and other numeric quantities) to Strings and vice versa. Choosing the correct technique can be confusing.

ints to Strings

To convert an int `iSource` to a String, there are at least three techniques:

```
"" + iSource;
```

or

```
new Integer(iSource).toString();
```

or

```
String.valueOf(iSource);    //  PREFERRED
```

Let's eliminate what we can. The first solution, which concatenates the int to an empty String, is just silly. It certainly involves less typing, but every time I have seen this technique, I have had to look twice to figure out what was going on.

In the second solution, creating an Integer first from the int and then using `toString()` to return its printable representation is considerably more expensive. It also declares a misleading intent: it performs a conversion by using a method intended to provide a printable representation of an object.

The class method `String.valueOf(int)` is the fastest implementation and provides the clearest code as well. The `valueOf()` method is also preferred as a generic solution for converting Object subclasses to strings, as it can handle the `null` object. The message `toString()`, on the other hand, will generate a NullPointerException if sent to the `null` object.

Strings to ints

For converting a String `sSource` to an int, there are also three solutions:

```
new Integer(sSource).intValue();
```

or

```
Integer.valueOf(sSource).intValue();
```

or

```
Integer.parseInt(sSource);   // PREFERRED
```

The first solution, creating a new Integer object and retrieving its `intValue()`, is a bit indirect.

Ideally, we'd like to have a technique for converting a String to an int that mirrors converting an int to a String. Consistency is always a noble goal. No dice, though — ints are not objects. If they were, we would have the method `int.valueOf(String)`. Unfortunately, we are stuck with Integer wrappers, as used in the second solution. We must first convert the String into an Integer, and then derive its `intValue()`. So the second solution also ends up being indirect.

It turns out that the `Integer.valueOf(String)` class method actually calls the `parseInt()` method to do its dirty work and then wraps the resultant int in an Integer object. By definition, unless some good optimization is going on, `parseInt()` will thus perform fastest. Use the third solution to convert Strings to ints.

QUERY METHOD

Answers the Question	How do you represent testing the property of an object?
Solution	Provide a method named like a query: Prefix the property to be tested with a form or variant of the verb "be": `isOpen()`, `hasDependents()`, `wasDeleted()`, for example. Return a `boolean` from the method.
Category	Behavioral
Related Patterns	Enumerated Constants

Many classes define attributes that hold simple true-false (boolean) values or enumerated constants in a short range. For example, in tracking employees you might have the need to store whether or not the employee works part-time, full-time, or flex-time, whether the employee is tax exempt, or whether the employee is a relative of the CEO.

The employee class might also need to provide testing that depends on combined criteria or calculated information. For instance, full-time status might be calculated from the number of hours worked per week.

Regardless of whether the object property is a state variable or a derived value, means of testing the condition should be the same. For booleans, this simply means that you return the attribute directly:

```
public boolean wasHiredByCEO()
{
   return wasHiredByCEO;
}
```

For enumerated data, the instance variable representing the attribute should not be directly exposed to client developers. Instead, provide a QUERY METHOD for each possible state of the instance variable. For example:

```
public boolean isFullTime()
{
   return status == FULL_TIME;
}

public boolean isPartTime()
{
   return status == PART_TIME;
}
```

These QUERY METHODS should be named starting with a form of the verb "be" — something like "is" or "was." For example:

```
isFullTime(), isPartTime(), wasHiredByCEO()
```

These method names also improve client code readability:

```
if (employee.isFullTime())
```

You should provide converse testing methods if possible, especially for true boolean attributes. For example, if you store a boolean attribute `isTaxExempt`, you should provide methods `isTaxExempt()` and `isTaxable()`. Whenever possible, be positive about things. Negativity is not good for the spirit and in general is more difficult to understand. Thus for a CD player application you might have methods `isTrayOpen()` and `isTrayClosed()` (instead of `isTrayNotOpen()`) to represent the single boolean attribute `isTrayOpen`.

If you have a domain that spans more than two or three values, or if you know that the range of your domain is likely to expand, use the ENUMERATED CONSTANTS pattern instead. For example, the employee status could be represented by many values. You might implement a subclass that provides constants for full-time, part-time, flex-time, and shift-time employee status values. Refer to ENUMERATED CONSTANTS in Chapter 3, "State Patterns," for more information on this option.

COMPARING METHOD

Answers the Question	How do you order objects with respect to each other?
Solution	Implement the Comparable interface within the objects to be ordered and call Collections.sort (List). Use Collections.sort (List, Comparator) if the elements are to be sorted in a non-natural sequence.
Category	Behavioral
Related Patterns	Equality Method

One feature that was sorely missed from Java — until version 2 — is built-in sorts. How disappointing. Coming from other languages, it's been more than a decade since I had to write my own sort. Even C libraries provide the `qsort()` function.

Version 2 provides a merge sort as part of its Collections Framework. You may wonder why quick sort, the ever-popular sort of choice, wasn't included. The reason quick sort is so prevalent is that it is very simple to write (almost as easy to write as the justly maligned bubble sort) and gives O(n log n) performance in most cases. However, in the worst case, when the data is already sorted, performance goes to $O(n^2)$. Merge sort is not trivial to write, but it is a stable[4] sort that is guaranteed to provide O(n log n) performance.

To sort in Java, either you must initialize the sort with a Comparator object, or the objects being sorted must implement the Comparable interface. Your sortable objects (usually your domain objects) should always define a natural sorting order — the specific order in which the objects are most often required. For a set of employees, this sort order is typically by the employees' full names. To implement a natural sorting order, declare that your class will implement the Comparable interface and provide its required `compareTo()` method.

The `compareTo()` method takes a single parameter, the object to which `this` is to be compared. `compareTo()` should return one of three domains of values. For the example `x.compareTo(y)`:

If	Return
x < y	a negative integer
x == y	0
x > y	a positive integer

4 A stable sort does not reorder equal elements, which is important when you are sorting the same list repeatedly using different attributes.

There are also three important requirements that this comparison method must meet. The JDK states the requirements as following:

JDK Terms	In English, Please
sgn(x.compareTo(y)) == -sgn(y.compareTo(x)) for all x and y	If x is less than y, then y must be greater than x, and vice versa. If x is equal to y, then y equals x must also be true.
x.compareTo(y) > 0 and y.compareTo(z) > 0 implies x.compareTo(z) > 0	If x is greater than y and y is greater than z, then x must be greater than z.
x.compareTo(y) == 0 implies sgn(x.compareTo(z)) == sgn(y.compareTo(z)), for all z	If x is equal to y, then the comparisons between x and any other object return the same result as the comparison between y and the same object.

The JDK also highly recommends that if `x.compareTo(y)` returns 0, then `x.equals(y)` returns true. If this is not the case, you probably should define a Comparator object instead. Comparators are objects used to provide special case ordering for a collection. A comparator must implement the `compare(Object, Object)` method, which works the same as the `compareTo()` method except that the objects to be compared are both passed as arguments. In most cases, you want to provide an `equals()` method for your Comparator. Otherwise, the Comparator uses the default implementation of `equals()` — object identity — to compare elements.

REVERSING METHOD

Answers the Question	How do you code a smooth flow of messages?
Solution	Add a new method to the class of a parameter. The new method takes the original receiver as a parameter and sends a "reversing" message back to the original receiver with `this` as a parameter.
Category	Behavioral
Related Patterns	None

One of the nice features about Java is that it allows for message chaining — using the resultant object of one message send as the receiver to a subsequent message send:

```
iterator.next().toString().toLowerCase();
```

This line of code sends the message `next()` to an instance variable presumably containing an Iterator object; the result is an Object. This resultant object is sent the message `toString()`, which returns a string. This string finally gets sent the message `toLowerCase()` and returns another string.

Of course, this is also a not-so-nice feature if it is abused. Sending long chains of messages can make it difficult to understand what is going on. Breaking up these chains with appropriate uses of EXPLAINING TEMPORARY VARIABLE can go a long way toward making your code clear in intent.

What if you want to send a series of messages to the same receiving object? Java does not provide special syntax for this. You must explicitly code multiple statements to achieve this effect, known as cascading.

Listing 1.13 ***Nice-Looking Cascade***

```
list.add("string 1");
list.add("second string");
list.add("3rd string");
list.remove("second string");
```

With the help of the text being left-aligned, as demonstrated in **Listing 1.13**, it is clear what is going on in this code. The code has a nice flow to it and is understandable at a glance. While many developers discount the value of visual considerations in code, it is extremely valuable to make your code as readable as possible. Humans generally consume things in chunks of a reasonable size; the smaller a chunk is and the more organized it is, the quicker it is interpreted by the human brain. If I throw nine coins on the floor and they fall in a random pattern, it will take you longer to count them than if they were in three rows of three.

Code is the same way. With larger methods — perhaps 10 or more lines — it will take longer to comprehend the code unless it makes judicious use of white space. Early BASIC coders would cram as much code on a single line as possible. No one with any sense does this any more; We all are good little coders because we put each statement on a separate line. You should strive to break up methods into understandable chunks, or better yet, into separate methods.

REVERSING METHOD is a visual enhancing pattern that is almost a formatting pattern. But because it involves altering behavior, it remains in this chapter. REVERSING METHOD helps you keep that flow of cascading messages looking pretty.

Occasionally you will be coding a cascading series of statements and find out that your clean flow has been rudely interrupted by a message send that has to go to a different object. **Listing 1.14** gives an example of this unfortunate code.

Listing 1.14 *Not-as-Nice-Looking Code*

```
canvas.moveTo(3, 4);
canvas.drawLineTo(7, 11);
bulletArt1.drawOn(canvas);
canvas.drawLineTo(12, 13);
bulletArt2.drawOn(canvas);
```

Using REVERSING METHOD, you first implement a `draw()` method in the class that created the canvas object. Then you simply *reverse* things and call the original `drawOn()` method, passing `this` as the parameter. The code is shown in **Listing 1.15**.

Listing 1.15 *REVERSING METHOD Example*

```
public void draw(Art art)
{
   art.drawOn(this);
}
```

Listing 1.16 shows the clean, concise cascade that is now possible, thanks to REVERSING METHOD.

Listing 1.16 *The Improved Cascade*

```
canvas.moveTo(3, 4)
canvas.drawLineTo(7, 11);
canvas.draw(bulletArt1);
canvas.drawLineTo(12, 13);
canvas.draw(bulletArt2);
```

If you want to do this with Java system classes, such as the Graphics class, you will need to create your own specialized subclasses so you can make the necessary modifications.

METHOD OBJECT

Answers the Question	How do you code a method where many lines of code share many arguments and temporary variables?
Solution	Define an inner class named after the method. Declare an instance variable in the class for each temporary variable in the original method; pass the temporary variables into the class via a single constructor. Define a method `compute()`, which triggers the process defined in the original method. Apply COMPOSED METHOD within the METHOD OBJECT.
Category	Behavioral
Related Patterns	Composed Method Parameter Object Collecting Temporary Variable

Occasionally you will come across a very long, very ugly (VLVU) method that uses a good number of stack variables and parameters as integral parts of its computation. If you attempt to apply COMPOSED METHOD to the method, you end up needing to pass the original parameters plus the large number of stack variables from method to method as parameters. You no longer have a VLVU method, but instead you have VLVU method signatures. You typically do not want to define these variables as instance variables. Not only would they clutter the definition of the class unnecessarily, but they violate the principle of instance variables by representing something other than object state.

I have come across the need for METHOD OBJECT a few times. Most of the time I solved it with PARAMETER OBJECT. But METHOD OBJECT goes one step further than PARAMETER OBJECT. It not only encapsulates the variables needed in a separate object, it also encapsulates the behavior that represents the complex method.

An abbreviated example follows. You might not bother implementing METHOD OBJECT for such a short example. Extrapolate, though, and imagine that the following method has several hundred lines involving several more tasks and several more stack variables.

A portion of the VLVU method is presented in **Listing 1.17**.

Listing 1.17 *A VLVU Method*

```
private void calculatePay(Employee employee,
                          int payPeriod)
{
   int baseSalary = employee.getBaseSalary();
   int contractedRate =
          employee.getContractedRate();
   boolean isContracted = contractedRate > 0;
   int hoursWorked =
          employee.getHoursWorked(payPeriod);
   String state = employee.getState();
   List taxItemizations = new ArrayList();
   List preTaxDeductions = new ArrayList();
   List postTaxDeductions = new ArrayList();

   // calculate bi-weekly pay amount
   double basePay =
      isContracted ?
         contractedRate * hoursWorked :
         baseSalary / 26;

   // calculate FICA tax
   double baseFICA = basePay * .0751;
   double ytdFICA = employee.getYtdFICA();
   double maxFICA = Payroll.getMaxFICA();
   if (ytdFICA > maxFICA)
      ficaTax = 0.0;
   else
      if (ytdFICA + baseFICA > maxFICA())
         ficaTax = maxFICA() – ytdFICA();
      else
         ficaTax = baseFICA;
   taxItemizations.add(new LineItem("FICA Tax",
                                    ficaTax));

   // calculate local tax
   boolean hasLocalTax = isInTaxDistrict(employee);
   //  yadda yadda yadda....

   // calculate state tax

   // out of state adjustments?

   // calculate federal income tax

   //  calculate medicare tax

   //  pre-tax deductions
   //    code for 401K, health care contrib, etc.
   //  post-tax deductions
   //     code for stock purchase plans,
   //     meal plans, etc.
}
```

As you can see, this is potentially a very large method. Don't bet that payroll systems you come across will have cleaner code than this. Most payroll systems I have come across seem to take pride in their inscrutable nature.

The parameters `employee` and `payPeriod` will be required in most of the computations. The COLLECTING TEMPORARY VARIABLES `taxItemizations`, `preTaxDeductions`, and `postTaxDeductions` will each be required in many separate computations. These five variables would have to be passed around to all of the various methods that would result from applying COMPOSED METHOD.

Using METHOD OBJECT instead, create a new class PayCalculator named after the original method (`calculatePay()`). Define its instance variables and constructor as in **Listing 1.18**.

Listing 1.18 *Instance Variables and Constructor for METHOD OBJECT*

```
private int payPeriod;
private Employee employee;
private List taxItemizations = new ArrayList();
private List preTaxDeductions = new ArrayList();
private List postTaxDeductions = new ArrayList();

protected PayCalculator(Employee _employee,
                        int _payPeriod)
{
   employee = _employee;
   payPeriod = _payPeriod;
}
```

Next, create a method within PayCalculator called `compute()`, and copy into it the code from the original VLVU `calculatePay()` method. The declaration of the COLLECTING TEMPORARY VARIABLES can be eliminated, as they now are available as instance variables. **Listing 1.19** shows the results.

Listing 1.19 *METHOD OBJECT `compute()` Method*

```
public compute()
{
   int baseSalary = employee.getBaseSalary();
   int contractedRate =
          employee.getContractedRate();
   boolean isContracted = contractedRate > 0;
   int hoursWorked =
          employee.getHoursWorked(payPeriod);
   String state = employee.getState();

   // calculate bi-weekly pay amount
   double basePay =
      isContracted ?
          contractedRate * hoursWorked :
          baseSalary / 26;
   //  ... rest of the method here
}
```

The original `calculatePay()` method becomes the code in **Listing 1.20**.

Listing 1.20 *Cleaned-Up VLVU Method*

```
private void calculatePay(Employee employee,
                          int payPeriod)
{

   PayCalculator calculator =
      new PayCalculator(employee,
                        payPeriod);
   calculator.compute();
}
```

Once this is in place, the next-to-last step is to ensure that the `compute()` method works. Testing must show that it mirrors the original behavior of `calculatePay()`.

Finally, apply COMPOSED METHOD to clean up the `compute()` method in PayCalculator. Almost every chunk of code identified by individual comments in the original `calculatePay()` method would become its own method. The modified `compute()` method, along with a couple of its refactored methods, would look something like **Listing 1.21**. Once again, you must test to ensure that you have not changed the required behavior.

Listing 1.21 *After Applying COMPOSED METHOD*

```
public compute()
{
   double basePay = calculateBasePay();
   calculateFICA(basePay);
   if (isInTaxDistrict())
      calculateLocalTax(basePay);
   calculateStateTax(basePay);
   calculateOutOfStateAdjustments(basePay);
   calculateFederalIncomeTax(basePay);
   calculateMedicareTax(basePay);
   calculatePreTaxDeductions();
   calculatePostTaxDeductions();
}

private double calculateBasePay()
{
   int contractedRate =
          employee.getContractedRate();
   boolean isContracted = contractedRate > 0;
```

```
if (isContracted)
      return
         contractedRate *
         employee.getHoursWorked(payPeriod);
   else
      return employee.getBaseSalary() / 26.0;
}

private void calculateFICA(double basePay)
{
   double baseFICA = basePay * .0751;
   double ytdFICA = employee.getYtdFICA();
   double maxFICA = Payroll.getMaxFICA();
   if (ytdFICA > maxFICA)
      ficaTax = 0.0;
   else
      if (ytdFICA + baseFICA > maxFICA())
         ficaTax = maxFICA() – ytdFICA();
      else
         ficaTax = baseFICA;
   taxItemizations.add(new LineItem("FICA Tax",
                                    ficaTax));
}
```

PARAMETER OBJECT

Answers the Question	How do you code a method where many lines of code share many arguments and temporary variables?
Solution	Apply COMPOSED METHOD; use an inner class to store the parameters that need to be passed from method to method. Access the parameters within the PARAMETER OBJECT directly from the COMPOSED METHODS.
Category	Behavioral
Related Patterns	Method Object Composed Method

This pattern is the converse of METHOD OBJECT.

Again, assume you have a VLVU method that uses a good number of stack variables and parameters as integral parts of its computation. Instead of creating a METHOD OBJECT to contain the attributes as encapsulated instance variables, an alternate solution is to create a new class that, for all intents and purposes, is a C struct. It will contain only instance variables representing the essential data.

You will not need accessor methods in the PARAMETER OBJECT, because you will be accessing its instance variables directly. This is one of the few cases where directly accessing instance variables externally is OK. I know, I said *never* do this. Lesson learned — never say never.

Define an inner class that stores the parameters. **Listing 1.22** demonstrates this for the payroll example from METHOD OBJECT.

Listing 1.22 *PARAMETER OBJECT Example*

```
public class Payroll
{
   // ... other methods 'n' stuff ...

   private void calculatePay(Employee employee,
                             int payPeriod)
   {
       // ...original method code here ...
   }

(3)    class CalculatePayParms
       {
          int payPeriod;
          Employee employee;
          List taxItemizations;
          List preTaxDeductions;
```

```
      List postTaxDeductions;
   }

   //  ... other methods 'n' stuff ...
}
```

Note that the definition of the inner class CalculatePayParms goes right after `calculatePay()` (line [3]). Also note that the class is named by appending "Parms" to the method name.

Once the inner class has been declared, create an instance of it in `calculatePay()` and directly set its instance variables. This modification of `calculatePay()` is shown in **Listing 1.23**.

Listing 1.23 *Using the PARAMETER OBJECT*

```
private void calculatePay(Employee employee,
                          int payPeriod)
{
   ComputeParms parms = new ComputeParms();
   parms.taxItemizations = new ArrayList();
   parms.preTaxDeductions = new ArrayList();
   parms.postTaxDeductions = new ArrayList();
   parms.payPeriod = payPeriod;
   parms.employee = employee;
   //  ... rest of method code ...
}
```

Now you can apply COMPOSED METHOD to `calculatePay()`. For each method invoked, simply pass it the instance of ComputeParms. Within the composed methods, directly access the instance variables of ComputeParms. **Listing 1.24** shows `calculatePay()` and some associated method implementations after COMPOSED METHOD has been applied.

Listing 1.24 *After Applying COMPOSED METHOD*

```
public calculatePay()
{
   ComputeParms parms = new ComputeParms();
   parms.taxItemizations = new ArrayList();
   parms.preTaxDeductions = new ArrayList();
   parms.postTaxDeductions = new ArrayList();
   parms.payPeriod = payPeriod;
   parms.employee = employee;

   double basePay = calculateBasePay(parms);
   calculateFICA(parms, basePay);
   if (isInTaxDistrict(parms))
      calculateLocalTax(parms, basePay);
   calculateStateTax(parms, basePay);
   calculateOutOfStateAdjustments(parms, basePay);
```

(continued)

Listing 1.24 (cont.) *After Applying COMPOSED METHOD*

```
   calculateFederalIncomeTax(parms, basePay);
   calculateMedicareTax(parms, basePay);
   calculatePreTaxDeductions(parms);
   calculatePostTaxDeductions(parms);
}

private double calculateBasePay(ComputeParms parms)
{
   int contractedRate =
          parms.employee.getContractedRate();
   boolean isContracted = contractedRate > 0;
   if (isContracted)
      return
         contractedRate *
         parms.employee.getHoursWorked(
            parms.payPeriod);
   else
      return parms.employee.getBaseSalary() / 26.0;
}

private void calculateFICA(ComputeParms parms,
                           double basePay)
{
   double baseFICA = basePay * .0751;
   double ytdFICA = parms.employee.getYtdFICA();
   double maxFICA = Payroll.getMaxFICA();
   if (ytdFICA > maxFICA)
      ficaTax = 0.0;
   else
      if (ytdFICA + baseFICA > maxFICA())
         ficaTax = maxFICA() – ytdFICA();
      else
         ficaTax = baseFICA;
   parms.taxItemizations.add(
      new LineItem("FICA Tax", ficaTax));
}
```

When should you prefer METHOD OBJECT over PARAMETER OBJECT? Use METHOD OBJECT when the number of methods resulting from applying COMPOSED METHOD becomes excessive and clutters the original class definition. Otherwise, PARAMETER OBJECT should suffice.

DEBUG PRINTING METHOD

Answers the Question	How do you provide a printable representation of an object for debugging purposes?
Solution	Override `toString()` and have it return a concise string that uniquely describes the object.
Category	Behavioral
Related Patterns	None

Java provides the method `toString()` in Object. According to the JDK documentation, the purpose of this method is to return a string that "textually represents" this object.

The behavior that Object provides is to return a string with the class name followed by a hex representation of the object's hash code, for example, "Employee@fc825d21." This sort of information will not often be very useful, so `toString()` is one of the methods that you should override in your classes.

It is ill-advised to use `toString()` for purposes of representing objects within a user interface. The implementation of the method in the class Object should be a hint that this is not something your application users want to see. There is no guarantee that `toString()` will consistently provide a user-consumable representation.

In java.io.PrintWriter and java.io.PrintStream, the `toString()` method is indirectly called by the `print(Object)` method. The `print(Object)` method first calls `String.valueOf(Object)`, which in turn sends the message `toString()` to your object to get its printable representation. System.out, the standard output stream used often for debugging purposes,[5] is a PrintStream object.

Some IDEs, such as IBM's VisualAge for Java, use `toString()` as an integral part of the development environment. Instance inspectors send the message `toString()` to objects being inspected. This gives the developer an immediate, concise description of the object. The developer can avoid digging into the object to individually inspect its instance variables, which can mean lots of mouse clicks. Having appropriate debug print representations of objects is especially valuable when inspecting collections.

---

5 What year is this? A large number of developers I have come across who are doing enterprise or other large-scale Java development do not use one of the off-the-shelf IDE packages. Instead, it's usually a home-grown environment centered around a very customizable editor of choice. Debugging hearkens back to the days of inserting DISPLAY statements in COBOL code. Java is still a very immature development platform, and has a long way to evolve.

Use `toString()` to provide a short, concise debugging string. **Listing 1.25** shows an example `toString()` method for Employee.

Listing 1.25 *A `toString()` Method for Employee*

```
public String toString()
{
   return getName() + " (" + getSSN() + ")";
}
```

Example output from the `toString()` method in **Listing 1.25**:

```
Joseph Schmo (999-31-9999)
```

In addition, you may wish to provide the class information, especially if the debugging string is ambiguous within the context of a collection. For example, suppose you have an inheritance hierarchy with classes Management and Grunt inheriting from the superclass Employee. If you create a list of Employee objects, you will have no effective way to determine what type of employees are contained in the collection when you are debugging. **Listing 1.26** provides a more useful `toString()` method for your Employee class by also returning the object's class name.

Listing 1.26 *A `toString()` Method That Prints the Class Name*

```
public String toString()
{
   return
      getClass().getName() + " [" +
      getName() + " (" + getSSN() + ")]";
}
```

which would print something like:

```
Grunt [Joseph Schmo (999-31-9999)]
```

This is one possible convention. The important thing is to decide upon a standard and stick with it. In fact, the repetitive code in **Listing 1.26** cries out for COMPOSED METHOD. Factor out the common code to support whatever convention you choose. This will also help ensure consistency of presentation. For example, you might provide a method called `classTaggedString()` and pass it both the object (so the class name can be retrieved) and the string representing the object. A possible implementation is provided in **Listing 1.27**.

Listing 1.27 *`classTaggedString()` Example*

```
public String classTaggedString(Object object,
                                 String string)
{
   return object.getClass().getName() +
          " [" + string + "]";
}
```

The only question is where to put `classTaggedString()`. Many systems have a Debug class for managing debugging facilities; this method would probably work best as a class (static) method there.

With the above solution, your `toString()` method should look like the code in **Listing 1.28**.

Listing 1.28 *A `toString()` Method for Consistent Presentation*

```
public String toString()
{
   return
      Debug.classTaggedString(this,
                              getName() +
                              " (" + getSSN() +
                              ")");
}
```

METHOD COMMENT

Answers the Question	How do you comment methods?
Solution	Provide a developer-oriented comment, apart from the javadoc comment, at the beginning of a method, *only if necessary*. This comment should only communicate important information that is not obvious from the code. Refactor unclear code using other patterns, including COMPOSED METHOD.
Category	Behavioral
Related Patterns	Composed Method

Few things are more painful for the typical programmer than commenting code. More often than not, this exercise is performed well after the programming is complete, a practice generally looked upon with great disdain. But the reality is: Commenting after coding is complete requires that the programmer undergo the sometimes embarrassing task of having to understand his or her own code.

Sometimes the programmer is methodical enough to ensure that comments are inserted while he or she is coding. What this usually means is that the programmer is aware of the inscrutable nature of the code just written and figures that a comment would be prudent (in the off chance that the section of code in question has to be maintained).

Then there is the obligatory "commentator." This is either someone who is following departmental dogma to the letter or someone who perhaps should engage in a career as a legal copywriter. Virtually every line is commented, and every getter/setter method contains a comment. The volume of comments is almost always greater than the amount of code, as demonstrated in **Listing 1.29**.

Listing 1.29 ***Too Many Comments!***

```
/**
 * Copies one file, byte by byte, to another.
 * Quick & dirty but costly performance-wise
 * since IO operations are not buffered.
 * @param  from  String representing the filename
 *               to copy from
 * @param  to    String representing the filename
 *               to copy to
 * @exception    IOException if opening either
 *               input or output files fails
 */
```

```
public void fileCopy(String from, // copy-from file
                     String to)   // copy-to file
      throws IOException   // exception thrown if
                           // file ops. fail
{
   DataInputStream input =   // create input stream
     new DataInputStream(new FileInputStream(from));
   DataOutputStream output = // create output stream
     new DataOutputStream(new FileOutputStream(to));
   //  attempt to loop through the input file,
   //  writing byte-by-byte to the output file for
   //  each byte read from the input file.
   //  If an IO operation against either file
   //  fails, an IO exception is thrown.  But when
   //  the end of file is reached, an EOF exception
   //  is thrown.  This EOF exception is trapped
   //  and accepted as part of normal operation —-
   //  no error handling is done.  The files are
   //  subsequently closed.
   try  // trap read/write operation exceptions
   {
      while (true)  // loop infinitely
         output.writeByte(
        input.readByte()); // read & write
                           // the same byte
   }
   catch (EOFException e) {} // trap the inevitable
                             //  EOF exception
                             //  but do nothing
   finally     //  regardless of result
   {
      input.close();   //  close input file
      output.close();  // close output file
   }
}
```

Not only are most of the comments in code like **Listing 1.29** useless, they tend to clutter to the code, making it more difficult to read. The worst aspect of comments is that they are just that, plain English comments — one person's interpretation of what they think is going on. There is no way to completely enforce the accuracy of comments versus code. The result is that inaccurate comments are inserted into code, and accurate comments rapidly become outdated when code is modified.

Some automation tools exist. VisualWorks Smalltalk, for example, includes a class comment checker. This tool notes the class type for all instance variables declared in the class comment. Then the class comment checker ensures that the actual messages being sent to these instance variable objects match the declared type.

The Java solution is to provide a level of automation that can expose code information to people who do not necessarily read code. Javadoc stipulates a specific format for code commenting. If the developer correctly

follows this specification, javadoc can be run against the code to produce a nicely formatted, highly hypertext-oriented, indexed Web reference. Javasoft's JDK API specifications on the Web are themselves generated by javadoc.

Javadoc does some minor consistency checking, but for the most part, it just takes what it finds and spits it out in HTML format. If you forget to add an `@param` comment for a method parameter, javadoc doesn't complain at all. There are other problems with javadoc; in short, do not trust that javadoc will catch all documentation problems.

The real problem with javadoc, however, is that it is intended to be a tool for providing a published API specification. Yet many programmers also use it as the end-all for what comments a program should have. "If it isn't something javadoc can parse out, why bother including the comment?" But the premise of javadoc is to enhance user (client) understanding of a package, not to provide detailed documentation so a developer can maintain the contents of a package. Proper Javadoc comments will impart some information to a developer modifying the class, but only as a side effect of what they provide to a client *using* the class.

Javadoc standards *should* always be followed by the developer, but only as far as they document a package's public interface. Javadoc comments for private methods are unnecessary and only help bolster the misguided idea that they solve the code documentation problem.

So what is the solution to the documentation problem? Should every method contain a comment? Every block?

By adding an obligatory comment to every method, you have significantly increased maintenance time with possibly no gain. Programmers tend to be lazy about adding comments in the first place; why would they be any better at maintaining them? Far worse than no comment is an inaccurate comment.

Adding a comment to a getting method like the one in **Listing 1.30** is not only wasted effort, it is insulting.

Listing 1.30 ***Useless Accessor Comment***

```
public String getCustomerSSN()
{
   // return the customer's SSN
   return customerSSN;
}
```

Also, if the code needs to be changed to retrieve a generic ID instead of an SSN, I warrant that you will see code like that in **Listing 1.31** fairly often.

Listing 1.31 *Useless* and *Incorrect Accessor Comment*

```
public String getCustomerID()
{
(4)    // return the customer's SSN
   return customerID;
}
```

Now, the comment [4] implies something that is not necessarily true. What are valid things to comment?

- Confusing code: Code that is not obvious.
- Dependencies: A method requires that another method be executed first.
- To do: Code that is not complete.
- Change reason: Why code was changed in a method.

If you find yourself writing a large amount of the first type of comment — for confusing code — you need to apply more of the patterns in this book to clean it up. If you refactor your code appropriately, the amount of code that requires commenting should dwindle to well below 10%. **Listing 1.32** provides a very simple example of where a comment is necessary to explain what is being tested.

Listing 1.32 *Code That Requires a Comment*

```
if (ftpResponse.charAt(0) == '2')
   // ftp result in 200s = success
```

If this test is executed in multiple places throughout the code, the same comment will end up in all places as well. And of course, if the FTP result codes were to change, the code would have to change in all those places.

Using COMPOSED METHOD, create the new method in **Listing 1.33**.

Listing 1.33 *Factoring into a Method*

```
public boolean wasSuccessful(String ftpResponse)
{
   return ftpResponse.charAt(0) == '2';
}
```

The client (calling) code is simplified to what is shown in **Listing 1.34**.

Listing 1.34 *Look, Mom, No More Comment*

```
if (wasSuccessful(ftpResponse))
```

The end result is that the client code no longer needs a comment. Nor does the code in the `wasSuccessful()` method.

INTENTION-REVEALING METHOD NAME

Answers the Question	What should a method be named?
Solution	Name a method after what it does, not how it does it.
Category	Behavioral
Related Patterns	Composed Method Getting Method Setting Method Query Method

Method naming is one of the most critical parts of class design. Without appropriate public method names, clients will have no clue as to how to use the class. Without appropriate private method names, maintenance developers will spend much more time trying to understand a class.

If you use COMPOSED METHOD, you will have very short methods, each of which performs a single task that can be precisely defined in a few words. Usually the most appropriate method name will be fairly obvious. Generally it is a verb that describes the action being completed by the method — `transmit()`, `encode()`, and the like. But if your method does many different tasks, coming up with a good method name that declares what the method is doing can be rather difficult. You will find yourself with methods called `processData()` and `doLotsOfWork()`.

The bulk of your methods can be categorized as action methods, testing methods, and accessor methods. Action methods, as mentioned, should be verbs. Naming conventions for accessor (getting and setting) methods are described in GETTING METHOD and SETTING METHOD. For testing methods, use the naming convention described in QUERY METHOD.

Spell out the names of your methods. Avoid abbreviations, unless they are common in the business domain of your application. Instead of `getFld()`, call your method `getField()`. Instead of `fsetpos()`, use `setPosition()`. You will type your code once (you hope); it may be read hundreds of times. The seconds you save typing are not worth it. Strive for readability.

Simple Implementations

Providing reusable methods for a class is an exercise in determining how you think the class might be used. Often the class evolves as new uses are determined, and sometimes the methods do not reflect what they are accomplishing. For some extremely simple operations, you might not even provide methods to accomplish them directly.

If you are the developer of an object, looking at how client code uses your objects can be revealing. Let's say your Employee object is in heavy use. Looking at the client code, you find the following expression scattered throughout:

```
if (employee.getTerminationDate() != null)
```

The code determines whether or not an employee is terminated.

Even in this small amount of code, there are a few problems. First, it does not read well. "Not" logic is frowned upon by many (perhaps a bit more than is necessary) and generally forces you to think more about what is really going on. Thus a comment would be prudent for the code:

```
if (employee.getTerminationDate() != null)
   // employee is terminated
```

But by now you should know that the need for a comment often cries out for you to apply COMPOSED METHOD. Except in this case, the method should be factored out of client code and into your Employee class. If your class's client developers have read this book, it's to be hoped that they have come to you by now and insisted that you make the change.

Second, even though this code represents a very simple test, it depends on a specific implementation — the fact that a termination date is null if the employee is still on the payroll. What if the Employee class changes its implementation so that open dates are represented by any date in the year 9999?

Using INTENTION-REVEALING METHOD NAME, your method on Employee should be what is shown in **Listing 1.35**.

Listing 1.35 *INTENTION-REVEALING METHOD NAME Example 1*

```
public boolean isTerminated()
{
   return getTerminationDate() != null;
}
```

This is also a QUERY METHOD.

A similar reason to apply INTENTION-REVEALING METHOD NAME is when you are exposing an implementation detail via the method name. A Stack implementation needs to provide a method to return the top element. If the Stack is implemented with a Vector, `lastElement()` provides this functionality. This does not reveal intent, it reveals implementation. It also breaks encapsulation — What if I change my Stack to use a data structure where returning the first element is more efficient?

Use INTENTION-REVEALING METHOD NAME to remap the implementation to a more meaningful message. **Listing 1.36** provides the `top()` method for the stack example.

Listing 1.36 *INTENTION-REVEALING METHOD NAME Example 2*

```
public Object top()
{
   return lastElement();
}
```

One interesting use of INTENTION-REVEALING METHOD NAME in the JDK has been to support deprecated methods. The odd thing is that the methods to be ultimately phased out are the ones that have the actual implementation. For example, the java.awt.List component has deprecated the `clear()` method in favor of `removeAll()`. But if you look at the code, `removeAll()` ends up delegating to `clear()`, which is where the real work ends up getting done.

You would think things would be the other way around — that the deprecated method should be the one incurring the performance penalty of a second message send. The decision to delegate from the newer method to the older one was done for backward-compatibility reasons.