Matt Weisfeld

Fourth Edition

The Object-Oriented Thought Process

Developer's Library

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

The Object-Oriented Thought Process

Fourth Edition

Developer's Library

ESSENTIAL REFERENCES FOR PROGRAMMING PROFESSIONALS

Developer's Library books are designed to provide practicing programmers with unique, high-quality references and tutorials on the programming languages and technologies they use in their daily work.

All books in the *Developer's Library* are written by expert technology practitioners who are especially skilled at organizing and presenting information in a way that's useful for other programmers.

Key titles include some of the best, most widely acclaimed books within their topic areas:

PHP & MySQL Web Development
Luke Welling & Laura Thomson
ISBN 978-0-672-32916-6

MySQL
Paul DuBois
ISBN-13: 978-0-672-32938-8

Linux Kernel Development
Robert Love
ISBN-13: 978-0-672-32946-3

Python Essential Reference
David Beazley
ISBN-13: 978-0-672-32978-4

Programming in Objective-C
Stephen Kochan
ISBN-13: 978-0-672-32756-8

C++ Primer Plus
Stephen Prata
ISBN-13: 978-0321-77640-2

Developer's Library books are available at most retail and online bookstores, as well as by subscription from Safari Books Online at **safari.informit.com**

Developer's Library
informit.com/devlibrary

The Object-Oriented Thought Process

Fourth Edition

Matt Weisfeld

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

The Object-Oriented Thought Process, Fourth Edition

ISBN-13: 978-0-321-86127-6
ISBN-10: 0-321-86127-2

Library of Congress Cataloging-in-Publication data is on file.

First Printing March 2013

Trademarks

Warning and Disclaimer

Bulk Sales

Pearson offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact

U.S. Corporate and Government Sales

1-800-382-3419

corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact

International Sales

international@pearsoned.com

Acquisitions Editor
Mark Taber

Development Editor
Songlin Qiu

Managing Editor
Sandra Schroeder

Project Editor
Seth Kerney

Copy Editor
Barbara Hacha

Indexer
Brad Herriman

Proofreader
Sarah Kearns

Technical Reviewer
Jon Upchurch

Editorial Assistant
Vanessa Evans

Interior Designer
Gary Adair

Cover Designer
Chuti Prasertsith

Compositor
Bronkella Publishing LLC

Contents at a Glance

Table of Contents

About the Author

Matt Weisfeld is a college professor, software developer, and author based in Cleveland, Ohio. Prior to teaching college full time, he spent 20 years in the information technology industry as a software developer, entrepreneur, and adjunct professor. Weisfeld holds an MS in computer science and an MBA. Besides the first three editions of *The Object-Oriented Thought Process*, he has authored two other software development books and published many articles in magazines and journals, such as *developer.com*, *Dr. Dobb's Journal*, *The C/C++ Users Journal*, *Software Development Magazine*, *Java Report*, and the international journal *Project Management*.

Dedication

❖

To Sharon, Stacy, Stephanie, and Duffy

❖

Acknowledgments

As with the first three editions, this book required the combined efforts of many people. I would like to take the time to acknowledge as many of these people as possible, for without them, this book would never have happened.

First and foremost, I would like to thank my wife Sharon for all her help. Not only did she provide support and encouragement throughout this lengthy process, she is also the first line editor for all my writing.

I would also like to thank my mom and the rest of my family for their continued support.

It is hard to believe that the work on the first edition of this book began in 1998. For all these years, I have thoroughly enjoyed working with everyone at Pearson—on all four editions. Working with editors Mark Taber, Songlin Qiu, Barbara Hacha, and Seth Kerney has been a pleasure.

A special thanks goes to Jon Upchurch for his expertise with much of the code as well as the technical editing of the manuscript. Jon's insights into an amazing range of technical topics have been of great help to me.

I would also like to thank Donnie Santos for his insights into mobile and hybrid development, as well as Objective-C.

Finally, thanks to my daughters, Stacy and Stephanie, and my cat, Duffy, for always keeping me on my toes.

We Want to Hear from You!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

We welcome your comments. You can email or write to let us know what you did or didn't like about this book—as well as what we can do to make our books better.

Please note that we cannot help you with technical problems related to the topic of this book.

When you write, please be sure to include this book's title and author as well as your name and email address. We will carefully review your comments and share them with the author and editors who worked on the book.

Email: feedback@developers-library.info

Mail: Reader Feedback
Addison-Wesley Developer's Library
Pearson Education
800 East 96th Street
Indianapolis, IN 46240

Reader Services

Visit our website and register this book at informit.com/register for convenient access to any updates, downloads, or errata that might be available for this book.

Introduction

This Book's Scope

As the title suggests, this book is about the object-oriented (OO) thought process. Although choosing the theme and title of a book are important decisions, these decisions are not at all straightforward when dealing with a highly conceptual topic. Many books deal with one level or another of programming and object orientation. Several popular books cover topics including OO analysis, OO design, OO programming, design patterns, OO data (XML), the Unified Modeling Language (UML), OO Web development, OO Mobile development, various OO programming languages, and many other topics related to OO programming.

However, while poring over all these books, many people forget that all these topics are built on a single foundation: how you think in OO ways. Often, many software professionals, as well as students, dive into these books without taking the appropriate time and effort to *really* understand the design concepts behind the code.

I contend that learning OO concepts is not accomplished by learning a specific development method, a programming language, or a set of design tools. Doing things in an OO manner is, simply put, a way of thinking. This book is all about the OO thought process.

Separating the languages, development practices, and tools from the OO thought process is not an easy task. Often, people are introduced to OO concepts by diving headfirst into a programming language. For example, many years ago, a large number of C programmers were first introduced to object orientation by migrating directly to C++ before they were even remotely exposed to OO concepts. Other software professionals' first exposure to object orientation was in the context of presentations that included object models using UML—again, before they were even exposed directly to OO concepts. Even now, a couple of decades after the emergence of the Internet as a business platform, it is not unusual to see programming books and professional training materials defer OO concepts until later in the discussion.

It is important to understand the significant difference between learning OO concepts and programming in an OO language. This came into sharp focus for me well before I worked on the first edition of this book, when I read articles like Craig Larman's "What the UML Is—and Isn't." In this article, he states,

Unfortunately, in the context of software engineering and the UML diagramming language, acquiring the skills to read and write UML notation seems to sometimes be equated with skill in object-oriented analysis and design. Of course, this is not so, and the latter is much more important than the former. Therefore, I recommend seeking education and educational materials in which intellectual skill in object-oriented analysis and design is paramount rather than UML notation or the use of a case tool.

Thus, although learning a modeling language is an important step, it is much more important to learn OO skills first. Learning UML before fully understanding OO concepts is similar to learning how to read an electrical diagram without first knowing anything about electricity.

The same problem occurs with programming languages. As stated earlier, many C programmers moved into the realm of object orientation by migrating to C++ before being directly exposed to OO concepts. This would always come out in an interview. Many times, developers who claim to be C++ programmers are simply C programmers using C++ compilers. Even now, with languages such as C# .NET, VB .NET, Objective-C, and Java well established, a few key questions in a job interview can quickly uncover a lack of OO understanding.

Early versions of Visual Basic are not OO. C is not OO, and C++ was *developed* to be backward compatible with C. Because of this, it is quite possible to use a C++ compiler writing only C syntax while forsaking all of C++'s OO features. Objective-C was designed as an extension to the standard ANSI C language. Even worse, a programmer can use just enough OO features to make a program incomprehensible to OO and non-OO programmers alike.

Thus, it is of vital importance that while you're learning to use OO development environments, you first learn the fundamental OO concepts. Resist the temptation to jump directly into a programming language (such as Objective-C, VB .NET, C++, C# .NET, or Java) or a modeling language (such as UML), and instead take the time to learn the object-oriented thought process.

After programming in C for many years, I took my first Smalltalk class in the late 1980s. The company I was with at the time had determined that its software developers needed to learn this up-and-coming technology. The instructor opened the class by stating that the OO paradigm was a totally new way of thinking (*despite the fact that it has been around since the 60s*). He went on to say that although all of us were most likely very good programmers, about 10%–20% of us would never really grasp the OO way of doing things. If this statement is indeed true, it is most likely because some good programmers never take the time to make the paradigm shift and learn the underlying OO concepts.

What's New in the Fourth Edition

As stated often in this introduction, my vision for the first edition was to stick to the concepts rather than focus on a specific emerging technology. Although I still adhere to this goal for the second, third, and fourth editions, I have included chapters on several application topics that fit well with object-oriented concepts. Chapters 1–10 cover the fundamental object-oriented concepts, and Chapters 11–15 are focused on applying these concepts to some general object-oriented technologies. For example, Chapters 1–10 provide the foundation for a course on object-oriented fundamentals (such as encapsulation, polymorphism, inheritance, and the like), with Chapters 11–15 adding some practical applications.

For the fourth edition, I expanded on many of the topics of the previous editions. These revised and updated topics include coverage of the following:

- Mobile device development, which includes phone apps, mobile apps and mobile/web, hybrids, and so on
- Objective-C code examples to include the iOS environment

- Human-readable data interchange using XML and JSON
- Rendering and transformation of data using CSS, XSLT, and so on
- Web services, including Simple Object Access Protocol (SOAP), RESTful Web Services, and the like
- Client/server technologies and marshaling objects
- Persistent data and serializing objects
- Expanded code examples, for certain chapters, in Java, C# .NET, VB .NET, and Objective-C available online on the publisher's website

The Intended Audience

This book is a general introduction to fundamental OO concepts, with code examples to reinforce the concepts. One of the most difficult juggling acts was to keep the code conceptual while still providing a solid code base. The goal of this book is to enable a reader to understand the concepts and technology without having a compiler at hand. However, if you do have a compiler available, there is code to be executed and explored.

The intended audience includes business managers, designers, developers, programmers, project managers, and anyone who wants to gain a general understanding of what object orientation is all about. Reading this book should provide a strong foundation for moving to other books covering more advanced OO topics.

Of these more advanced books, one of my favorites is *Object-Oriented Design in Java*, by Stephen Gilbert and Bill McCarty. I really like the approach of the book and have used it as a textbook in classes I have taught on OO concepts. I cite *Object-Oriented Design in Java* often throughout this book, and I recommend that you graduate to it after you complete this one.

Other books that I have found very helpful include *Effective C++*, by Scott Meyers; *Classical and Object-Oriented Software Engineering*, by Stephen R. Schach; *Thinking in C++*, by Bruce Eckel; *UML Distilled*, by Martin Fowler; and *Java Design*, by Peter Coad and Mark Mayfield.

While teaching intro-level programming and web development classes to programmers at corporations and universities, it quickly became obvious to me that most of these programmers easily picked up the language syntax; however, these same programmers struggled with the OO nature of the language.

The Book's Approach

It should be obvious by now that I am a firm believer in becoming comfortable with the object-oriented thought process before jumping into a programming language or modeling language. This book is filled with examples of code and UML diagrams; however, you do not need to know a specific programming language or UML to read it. After all I have said about learning the concepts first, why is there so much Java, C# .NET, VB .NET, and Objective-C code, as well as so many UML diagrams? First, they are great for illustrating OO concepts. Second, they are vital to the OO process and should be addressed at an introductory level. The key is not to focus on Java, C# .NET, VB .NET, and Objective-C or UML, but to use them as aids in the understanding of the underlying concepts.

Note that I really like using UML class diagrams as a visual aid in understanding classes, and their attributes and methods. In fact, the class diagrams are the only component of UML that is used in this book. I believe that the UML class diagrams offer a great way to model the conceptual nature of object models. I continue to use object models as an educational tool to illustrate class design and how classes relate to one another.

The code examples in the book illustrate concepts such as loops and functions. However, understanding the code itself is not a prerequisite for understanding the concepts; it might be helpful to have a book at hand that covers specific languages' syntax if you want to get more detailed.

I cannot state too strongly that this book does *not* teach Java, C# .NET, VB .NET, Objective-C, or UML, all of which can command volumes unto themselves. It is my hope that this book will whet your appetite for other OO topics, such as OO analysis, object-oriented design, and OO programming.

This Book's Conventions

The following conventions are used in this book:

- Code lines, commands, statements, and any other code-related terms appear in a `monospace` typeface.
- Throughout the book, there are special sidebar elements, such as the following:

> **Tip**
>
> A Tip offers advice or shows you an easy way of doing something.

> **Note**
>
> A Note presents interesting information related to the discussion—a little more insight or a pointer to some new technique.

> **Caution**
>
> A Caution alerts you to a possible problem and gives you advice on how to avoid it.

Source Code Used in This Book

The sample code described throughout this book is available on the publisher's website. Go to informit.com/register and register your book for access to downloads.

3

Advanced Object-Oriented Concepts

Chapter 1, “Introduction to Object-Oriented Concepts,” and Chapter 2, “How to Think in Terms of Objects,” cover the basics of object-oriented (OO) concepts. Before we embark on our journey to learn some of the finer design issues relating to building an OO system, we need to cover a few more advanced OO concepts, such as constructors, operator overloading, and multiple inheritance. We also will consider error-handling techniques and the importance of understudying how scope applies to object-oriented design.

Some of these concepts might not be vital to understanding an OO design at a higher level, but they are necessary to anyone involved in the design and implementation of an OO system.

Constructors

Constructors may be a new concept for structured programmers. Although constructors are not normally used in non-OO languages such as COBOL, C, and Basic, the *struct*, which is part of C/C++, does include constructors. In the first two chapters, we alluded to these special methods that are used to *construct* objects. In some OO languages, such as Java and C#, constructors are methods that share the same name as the class. Visual Basic .NET uses the designation *New* and Objective-C uses the *init* keyword. As usual, we will focus on the concepts of constructors and not cover the specific syntax of all the languages. Let’s take a look at some Java code that implements a constructor.

For example, a constructor for the `Cabbie` class we covered in Chapter 2 would look like this:

```
public Cabbie(){
    /* code to construct the object */
}
```

The compiler will recognize that the method name is identical to the class name and consider the method a constructor.

> **Caution**
>
> Note that in this Java code (as with C# and C++), a constructor does not have a return value. If you provide a return value, the compiler will not treat the method as a constructor.

For example, if you include the following code in the class, the compiler will not consider this a constructor because it has a return value—in this case, an integer:

```
public int Cabbie(){
    /* code to construct the object */
}
```

This syntax requirement can cause problems because this code will compile but will not behave as expected.

When Is a Constructor Called?

When a new object is created, one of the first things that happens is that the constructor is called. Check out the following code:

```
Cabbie myCabbie = new Cabbie();
```

The `new` keyword creates a new instance of the `Cabbie` class, thus allocating the required memory. Then the constructor itself is called, passing the arguments in the parameter list. The constructor provides the developer the opportunity to attend to the appropriate initialization.

Thus, the code `new Cabbie()` will instantiate a `Cabbie` object and call the `Cabbie` method, which is the constructor.

What's Inside a Constructor?

Perhaps the most important function of a constructor is to initialize the memory allocated when the `new` keyword is encountered. In short, code included inside a constructor should set the newly created object to its initial, stable, safe state.

For example, if you have a counter object with an attribute called `count`, you need to set `count` to zero in the constructor:

```
count = 0;
```

> **Initializing Attributes**
>
> In structured programming, a routine named housekeeping (or initialization) is often used for initialization purposes. Initializing attributes is a common function performed within a constructor.

The Default Constructor

If you write a class and do not include a constructor, the class will still compile, and you can still use it. If the class provides no explicit constructor, a default constructor will be provided. It is important to understand that at least one constructor always exists, regardless of whether you write a constructor yourself. If you do not provide a constructor, the system will provide a default constructor for you.

Besides the creation of the object itself, the only action that a default constructor takes is to call the constructor of its superclass. In many cases, the superclass will be part of the language framework, like the `Object` class in Java. For example, if a constructor is not provided for the `Cabbie` class, the following default constructor is inserted:

```
public Cabbie(){
    super();
}
```

If you were to decompile the bytecode produced by the compiler, you would see this code. The compiler actually inserts it.

In this case, if `Cabbie` does not explicitly inherit from another class, the `Object` class will be the parent class. Perhaps the default constructor might be sufficient in some cases; however, in most cases, some sort of memory initialization should be performed. Regardless of the situation, it is good programming practice to always include at least one constructor in a class. If there are attributes in the class, it is always good practice to initialize them. Moreover, initializing variables is always a good practice when writing code, object-oriented or not.

> **Providing a Constructor**
>
> The general rule is that you should *always* provide a constructor, even if you do not plan to do anything inside it. You can provide a constructor with nothing in it and then add to it later. Although there is technically nothing wrong with using the default constructor provided by the compiler, for documentation and maintenance purposes, it is always nice to know exactly what your code looks like.

It is not surprising that maintenance becomes an issue here. If you depend on the default constructor and then subsequent maintenance adds another constructor, the default constructor is no longer created. In short, the default constructor is added only if you don't include any constructors. As soon as you include just one, the default constructor is not provided.

Using Multiple Constructors

In many cases, an object can be constructed in more than one way. To accommodate this situation, you need to provide more than one constructor. For example, let's consider the `Count` class presented here:

```
public class Count {

    int count;
```

```
    public Count(){
        count = 0;
    }
}
```

On the one hand, we want to initialize the attribute `count` to count to zero: We can easily accomplish this by having a constructor initialize `count` to zero as follows:

```
public Count(){
    count = 0;
}
```

On the other hand, we might want to pass an initialization parameter that allows `count` to be set to various numbers:

```
public Count (int number){
    count = number;
}
```

This is called *overloading a method* (overloading pertains to all methods, not just constructors). Most OO languages provide functionality for overloading a method.

Overloading Methods

Overloading allows a programmer to use the same method name over and over, as long as the signature of the method is different each time. The signature consists of the method name and a parameter list (see Figure 3.1).

Thus, the following methods *all* have different signatures:

```
public void getCab();


// different parameter list
public void getCab (String cabbieName);

// different parameter list
public void getCab (int numberOfPassengers);
```

Signature

public String getRecord(int key)

Signature = getRecord (int key)
method name + parameter list

Figure 3.1 The components of a signature.

Signatures

Depending on the language, the signature may or may not include the return type. In Java and C#, the return type is not part of the signature. For example, the following methods would conflict even though the return types are different:

```
public void getCab (String cabbieName);
public int getCab (String cabbieName);
```

The best way to understand signatures is to write some code and run it through the compiler.

By using different signatures, you can construct objects differently depending on the constructor used. This functionality is very helpful when you don't always know ahead of time how much information you have available. For example, when creating a shopping cart, customers may already be logged in to their account (and you will have all of their information). On the other hand, a totally new customer may be placing items in the cart with no account information available at all. In each case, the constructor would initialize differently.

Using UML to Model Classes

Let's return to the database reader example we used earlier in Chapter 2. Consider that we have two ways we can construct a database reader:

- Pass the name of the database and position the cursor at the beginning of the database.
- Pass the name of the database and the position within the database where we want the cursor to position itself.

Figure 3.2 shows a class diagram for the `DataBaseReader` class. Note that the diagram lists two constructors for the class. Although the diagram shows the two constructors, without the parameter list, there is no way to know which constructor is which. To distinguish the constructors, you can look at the corresponding code in the `DataBaseReader` class listed next.

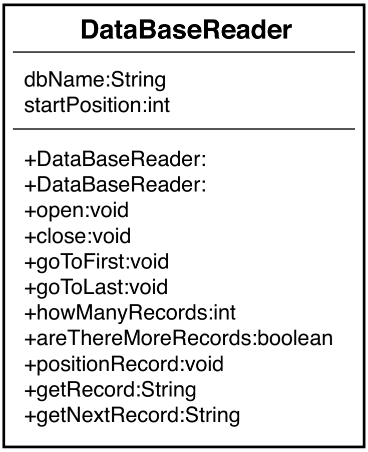

Figure 3.2 The `DataBaseReader` class diagram.

> **No Return Type**
> Notice that in this class diagram, the constructors do not have a return type. All other methods besides constructors must have return types.

Here is a code segment of the class that shows its constructors and the attributes that the constructors initialize (see Figure 3.3):

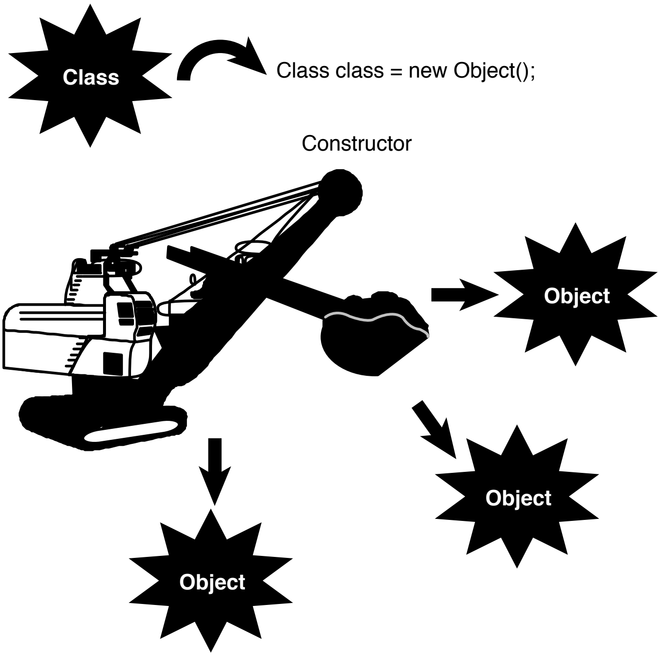

Figure 3.3 Creating a new object.

```
public class DataBaseReader {

    String dbName;
    int startPosition;

    // initialize just the name
    public DataBaseReader (String name){
        dbName = name;
      startPosition = 0;
    };

    // initialize the name and the position
    public DataBaseReader (String name, int pos){
```

```
        dbName = name;
        startPosition = pos;
    };

    .. // rest of class
}
```

Note how `startPosition` is initialized in both cases. If the constructor is not passed the information via the parameter list, it is initialized to a default value, such as 0.

How the Superclass Is Constructed

When using inheritance, you must know how the parent class is constructed. Remember that when you use inheritance, you are inheriting everything about the parent. Thus, you must become intimately aware of all the parent's data and behavior. The inheritance of an attribute is fairly obvious. However, how a constructor is inherited is not as obvious. After the `new` keyword is encountered and the object is allocated, the following steps occur (see Figure 3.4):

1. Inside the constructor, the constructor of the class's superclass is called. If there is no explicit call to the superclass constructor, the default is called automatically; however, you can see the code in the bytecodes.
2. Each class attribute of the object is initialized. These are the attributes that are part of the class definition (instance variables), not the attributes inside the constructor or any other method (local variables). In the `DataBaseReader` code presented earlier, the integer `startPosition` is an instance variable of the class.
3. The rest of the code in the constructor executes.

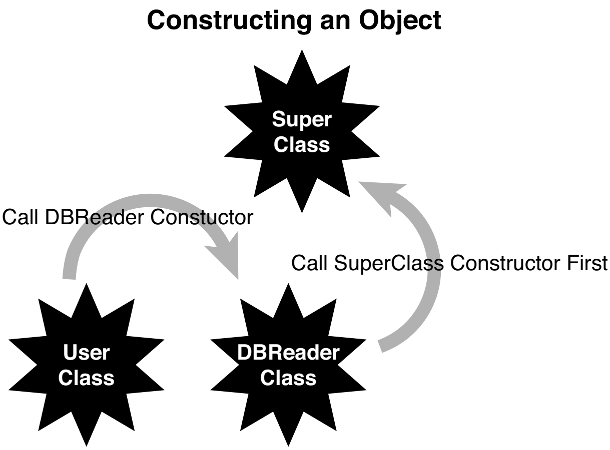

Figure 3.4 Constructing an object.

The Design of Constructors

As we have already seen, when designing a class, it is good practice to initialize all the attributes. In some languages, the compiler provides some sort of initialization. As always, don't count on the compiler to initialize attributes! In Java, you cannot use an attribute until it is initialized. If the attribute is first set in the code, make sure that you initialize the attribute to some valid condition—for example, set an integer to zero.

Constructors are used to ensure that the application is in a stable state (I like to call it a "safe" state). For example, initializing an attribute to zero, when it is intended for use as a denominator in a division operation, might lead to an unstable application. You must take into consideration that a division by zero is an illegal operation. Initializing to zero is not always the best policy.

During the design, it is good practice to identify a stable state for all attributes and then initialize them to this stable state in the constructor.

Error Handling

It is extremely rare for a class to be written perfectly the first time. In most, if not all, situations, things *will* go wrong. Any developer who does not plan for problems is courting danger.

Assuming that your code has the capability to detect and trap an error condition, you can handle the error in several ways: On page 223 of their book *Java Primer Plus*, Tyma, Torok, and Downing (9781571690623) state that there are three basic solutions to handling problems that are detected in a program: fix it, ignore the problem by squelching it, or exit the runtime in some graceful manner. On page 139 of their book *Object-Oriented Design in Java* (978-1571691347), Gilbert and McCarty expand on this theme by adding the choice of throwing an exception:

- Ignore the problem—not a good idea!
- Check for potential problems and abort the program when you find a problem.
- Check for potential problems, catch the mistake, and attempt to fix the problem.
- Throw an exception. (Often this is the preferred way to handle the situation.)

These strategies are discussed in the following sections.

Ignoring the Problem

Simply ignoring a potential problem is a recipe for disaster. And if you are going to ignore the problem, why bother detecting it in the first place? It is obvious that you should not ignore any known problem. The primary directive for all applications is that the application should never crash. If you do not handle your errors, the application will eventually terminate ungracefully or continue in a mode that can be considered an unstable state. In the latter case, you might not even know you are getting incorrect results, and that can be much worse than a program crash.

Checking for Problems and Aborting the Application

If you choose to check for potential problems and abort the application when a problem is detected, the application can display a message indicating that a problem exists. In this case, the application gracefully exits, and the user is left staring at the computer screen, shaking her head and wondering what just happened. Although this is a far superior option to ignoring the problem, it is by no means optimal. However, this does allow the system to clean up things and put itself in a more stable state, such as closing files and forcing a system restart.

Checking for Problems and Attempting to Recover

Checking for potential problems, catching the mistake, and attempting to recover is a far superior solution than simply checking for problems and aborting. In this case, the problem is detected by the code, and the application attempts to fix itself. This works well in certain situations. For example, consider the following code:

```
if (a == 0)
    a=1;

c = b/a;
```

It is obvious that if the conditional statement is not included in the code, and a zero makes its way to the divide statement, you will get a system exception because you cannot divide by zero. By catching the exception and setting the variable `a` to `1`, at least the system will not crash. However, setting `a` to `1` might not be a proper solution because the result would be incorrect. The better solution would be to prompt the user to reenter the proper input value.

> **A Mix of Error-Handling Techniques**
>
> Despite the fact that this type of error handling is not necessarily object-oriented in nature, I believe that it has a valid place in OO design. Throwing an exception (discussed in the next section) can be expensive in terms of overhead. Thus, although exceptions may be a valid design choice, you will still want to consider other error-handling techniques (even tried-and-true structured techniques), depending on your design and performance needs.

Although the error-checking techniques mentioned previously are preferable to doing nothing, they still have a few problems. It is not always easy to determine where a problem first appears. And it might take a while for the problem to be detected. In any event, it is beyond the scope of this book to explain error handling in great detail. However, it is important to design error handling into the class right from the start, and often the operating system itself can alert you to problems that it detects.

Throwing an Exception

Most OO languages provide a feature called *exceptions*. In the most basic sense, exceptions are unexpected events that occur within a system. Exceptions provide a way to detect problems and then handle them. In Java, C#, C++, Objective-C, and Visual Basic, exceptions are handled by the keywords `catch` and `throw`. This might sound like a baseball game, but the key concept here is that a specific block of code is written to handle a specific exception. This solves the problem of trying to figure out where the problem started and unwinding the code to the proper point.

Here is the structure for a Java `try/catch` block:

```
try {

    // possible nasty code

} catch(Exception e) {

    // code to handle the exception
}
```

If an exception is thrown within the `try` block, the `catch` block will handle it. When an exception is thrown while the block is executing, the following occurs:

1. The execution of the `try` block is terminated.
2. The `catch` clauses are checked to determine whether an appropriate `catch` block for the offending exception was included. (There might be more than one `catch` clause per `try` block.)
3. If none of the `catch` clauses handles the offending exception, it is passed to the next higher-level `try` block. (If the exception is not caught in the code, the system ultimately catches it, and the results are unpredictable—that is, an application crash.)
4. If a `catch` clause is matched (the first match encountered), the statements in the `catch` clause are executed.
5. Execution then resumes with the statement following the `try` block.

Suffice it to say that exceptions are an important advantage for OO programming languages. Here is an example of how an exception is caught in Java:

```
try {

    // possible nasty code
    count = 0;
    count = 5/count;

} catch(ArithmeticException e) {

    // code to handle the exception
    System.out.println(e.getMessage());
    count = 1;

}
System.out.println("The exception is handled.");
```

Exception Granularity

You can catch exceptions at various levels of granularity. You can catch all exceptions or check for specific exceptions, such as arithmetic exceptions. If your code does not catch an exception, the Java runtime will—and it won't be happy about it!

In this example, the division by zero (because `count` is equal to `0`) within the `try` block will cause an arithmetic exception. If the exception was generated (thrown) outside a `try` block, the program would most likely have been terminated (crashed). However, because the exception was thrown within a `try` block, the `catch` block is checked to see whether the specific exception (in this case, an arithmetic exception) was planned for. Because the `catch` block contains a check for the arithmetic exception, the code within the `catch` block is executed, thus setting `count` to `1`. After the `catch` block executes, the `try/catch` block is exited, and the message `The exception is handled.` appears on the Java console. The logical flow of this process is illustrated in Figure 3.5.

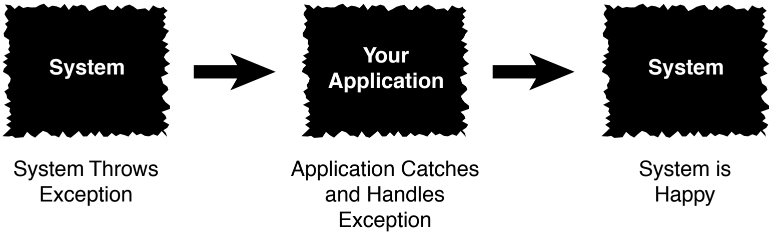

Figure 3.5 Catching an exception.

If you had not put `ArithmeticException` in the `catch` block, the program would likely have crashed. You can catch all exceptions by using the following code:

```
try {

    // possible nasty code

} catch(Exception e) {

    // code to handle the exception
}
```

The `Exception` parameter in the `catch` block is used to catch any exception that might be generated within a `try` block.

Bulletproof Code

It's a good idea to use a combination of the methods described here to make your program as bulletproof to your user as possible.

The Importance of Scope

Multiple objects can be instantiated from a single class. Each of these objects has a unique identity and state. This is an important point. Each object is constructed separately and is allocated its own separate memory. However, some attributes and methods may, if properly declared, be shared by all the objects instantiated from the same class, thus sharing the memory allocated for these class attributes and methods.

> **A Shared Method**
>
> A constructor is a good example of a method that is shared by all instances of a class.

Methods represent the behaviors of an object; the state of the object is represented by attributes. There are three types of attributes:

- Local attributes
- Object attributes
- Class attributes

Local Attributes

Local attributes are owned by a specific method. Consider the following code:

```
public class Number {

    public method1() {
        int count;

    }

    public method2() {

    }

}
```

The method `method1` contains a local variable called `count`. This integer is accessible only inside `method1`. The method `method2` has no idea that the integer `count` even exists.

At this point, we introduce a very important concept: scope. Attributes (and methods) exist within a particular scope. In this case, the integer count exists within the scope of `method1`. In Java, C#, C++ and Objective-C, scope is delineated by curly braces ({}). In the `Number` class, there are several possible scopes—just start matching the curly braces.

The class itself has its own scope. Each instance of the class (that is, each object) has its own scope. Both `method1` and `method2` have their own scopes as well. Because `count` lives within `method1`'s curly braces, when `method1` is invoked, a copy of `count` is created. When `method1` terminates, the copy of `count` is removed.

For some more fun, look at this code:

```
public class Number {

    public method1() {
        int count;
    }

    public method2() {
        int count;
    }

}
```

This example has two copies of an integer `count` in this class. Remember that `method1` and `method2` each has its own scope. Thus, the compiler can tell which copy of `count` to access simply by recognizing which method it is in. You can think of it in these terms:

```
method1.count;

method2.count;
```

As far as the compiler is concerned, the two attributes are easily differentiated, even though they have the same name. It is almost like two people having the same last name, but based on the context of their first names, you know that they are two separate individuals.

Object Attributes

In many design situations, an attribute must be shared by several methods within the same object. In Figure 3.6, for example, three objects have been constructed from a single class. Consider the following code:

```
public class Number {

    int count;    // available to both method1 and method2

    public method1() {
        count = 1;
    }

    public method2() {
        count = 2;
    }

}
```

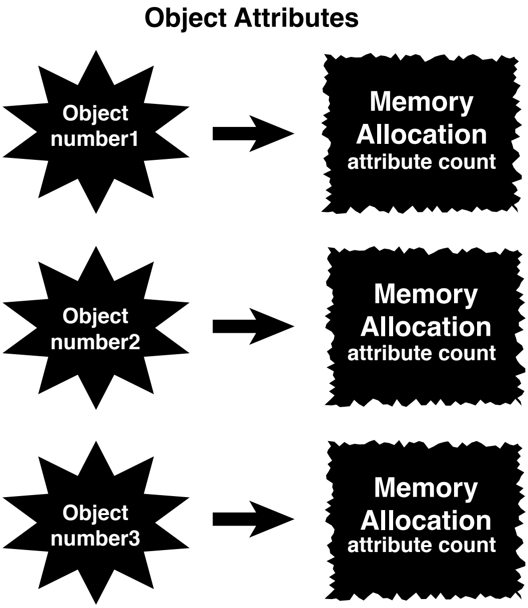

Figure 3.6 Object attributes.

Note here that the class attribute count is declared outside the scope of both `method1` and `method2`. However, it is within the scope of the class. Thus, `count` is available to both `method1` and `method2`. (Basically, all methods in the class have access to this attribute.) Notice that the code for both methods is setting `count` to a specific value. There is only one copy of `count` for the entire object, so both assignments operate on the same copy in memory. However, this copy of `count` is not shared between different objects.

To illustrate, let's create three copies of the `Number` class:

```
Number number1 = new Number();
Number number2 = new Number();
Number number3 = new Number();
```

Each of these objects—`number1`, `number2`, and `number3`—is constructed separately and is allocated its own resources. There are three separate instances of the integer `count`. When `number1` changes its attribute `count`, this in no way affects the copy of `count` in object `number2` or object `number3`. In this case, integer `count` is an *object attribute*.

You can play some interesting games with scope. Consider the following code:

```
public class Number {

    int count;
```

```
    public method1() {
        int count;
    }

    public method2() {
        int count;
    }

}
```

In this case, three totally separate memory locations have the name of `count` for each object. The object owns one copy, and `method1()` and `method2()` each have their own copy.

To access the object variable from within one of the methods, say `method1()`, you can use a pointer called `this` in the C-based languages:

```
public method1() {
        int count;

        this.count = 1;
}
```

Notice that some code looks a bit curious:

```
this.count = 1;
```

The selection of the word `this` as a keyword is perhaps unfortunate. However, we must live with it. The use of the `this` keyword directs the compiler to access the object variable `count` and not the local variables within the method bodies.

> **Note**
> The keyword `this` is a reference to the current object.

Class Attributes

As mentioned earlier, it is possible for two or more objects to share attributes. In Java, C#, C++ and Objective-C, you do this by making the attribute *static*:

```
public class Number {

    static int count;

    public method1() {
    }

}
```

By declaring `count` as static, this attribute is allocated a single piece of memory for all objects instantiated from the class. Thus, all objects of the class use the same memory location for `count`. Essentially, each class has a single copy, which is shared by all objects of that class (see Figure 3.7). This is about as close to global data as we get in OO design.

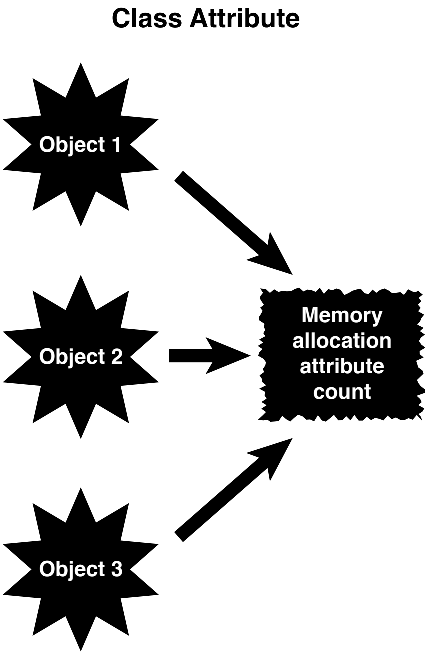

Figure 3.7 Class attributes.

There are many valid uses for class attributes; however, you must be aware of potential synchronization problems. Let's instantiate two `Count` objects:

```
Count Count1 = new Count();
Count Count2 = new Count();
```

For the sake of argument, let's say that the object `Count1` is going merrily about its way and is using `count` as a means to keep track of the pixels on a computer screen. This is not a problem until the object `Count2` decides to use attribute `count` to keep track of sheep. The instant that `Count2` records its first sheep, the data that `Count1` was saving is lost.

Operator Overloading

Some OO languages allow you to overload an operator. C++ is an example of one such language. Operator overloading allows you to change the meaning of an operator. For example, when most people see a plus sign, they assume it represents addition. If you see the equation

```
X = 5 + 6;
```

you expect that `X` would contain the value `11`. And in this case, you would be correct.

However, at times a plus sign could represent something else. For example, in the following code:

```
String firstName = "Joe", lastName = "Smith";

String Name = firstName + " " + lastName;
```

You would expect that `Name` would contain `Joe Smith`. The plus sign here has been overloaded to perform string concatenation.

> **String Concatenation**
>
> *String concatenation* occurs when two separate strings are combined to create a new, single string.

In the context of strings, the plus sign does not mean addition of integers or floats, but concatenation of strings.

What about matrix addition? You could have code like this:

```
Matrix a, b, c;

c = a + b;
```

Thus, the plus sign now performs matrix addition, not addition of integers or floats.

Overloading is a powerful mechanism. However, it can be downright confusing for people who read and maintain code. In fact, developers can confuse themselves. To take this to an extreme, it would be possible to change the operation of addition to perform subtraction. Why not? Operator overloading allows you to change the meaning of an operator. Thus, if the plus sign were changed to perform subtraction, the following code would result in an `X` value of `-1`:

```
x = 5 + 6;
```

More recent OO languages like Java, .NET, and Objective-C do not allow operator overloading.

Although these languages do not allow the option of overloading operators, the languages themselves do overload the plus sign for string concatenation, but that's about it. The designers of Java must have decided that operator overloading was more of a problem than it was worth. If you must use operator overloading in C++, take care by documenting and commenting properly not to confuse the people who will use the class.

Multiple Inheritance

We cover inheritance in much more detail in Chapter 7, "Mastering Inheritance and Composition." However, this is a good place to begin discussing multiple inheritance, which is one of the more powerful and challenging aspects of class design.

As the name implies, *multiple inheritance* allows a class to inherit from more than one class. In practice, this seems like a great idea. Objects are supposed to model the real world, are they not? And many real-world examples of multiple inheritance exist. Parents are a good example of multiple inheritance. Each child has two parents—that's just the way it is. So it makes sense that you can design classes by using multiple inheritance. In some OO languages, such as C++, you can.

However, this situation falls into a category similar to operator overloading. Multiple inheritance is a very powerful technique, and in fact, some problems are quite difficult to solve without it. Multiple inheritance can even solve some problems quite elegantly. However, multiple inheritance can significantly increase the complexity of a system, both for the programmer and the compiler writers.

As with operator overloading, the designers of Java, .NET, and Objective-C decided that the increased complexity of allowing multiple inheritance far outweighed its advantages, so they eliminated it from the language. In some ways, the Java, .NET, and Objective-C language construct of interfaces compensates for this; however, the bottom line is that Java, .NET, and Objective-C do not allow conventional multiple inheritance.

> **Behavioral and Implementation Inheritance**
>
> Interfaces are a mechanism for behavioral inheritance, whereas abstract classes are used for implementation inheritance. The bottom line is that interface language constructs provide behavioral interfaces, but no implementation, whereas abstract classes may provide both interfaces and implementation. This topic is covered in great detail in Chapter 8, "Frameworks and Reuse: Designing with Interfaces and Abstract Classes."

Object Operations

Some of the most basic operations in programming become more complicated when you're dealing with complex data structures and objects. For example, when you want to copy or compare primitive data types, the process is quite straightforward. However, copying and comparing objects is not quite as simple. On page 34 of his book *Effective C++*, Scott Meyers devotes an entire section to copying and assigning objects.

> **Classes and References**
>
> The problem with complex data structures and objects is that they might contain references. Simply making a copy of the reference does not copy the data structures or the object that it references. In the same vein, when comparing objects, simply comparing a pointer to another pointer only compares the references—not what they point to.

The problems arise when comparisons and copies are performed on objects. Specifically, the question boils down to whether you follow the pointers. Regardless, there should be a way to copy an object. Again, this is not as simple as it might seem. Because objects can contain references, these reference trees must be followed to do a valid copy (if you truly want to do a deep copy).

Deep Versus Shallow Copies

A *deep copy* occurs when all the references are followed and new copies are created for all referenced objects. Many levels might be involved in a deep copy. For objects with references to many objects, which in turn might have references to even more objects, the copy itself can create significant overhead. A *shallow copy* would simply copy the reference and not follow the levels. Gilbert and McCarty have a good discussion about what shallow and deep hierarchies are on page 265 of *Object-Oriented Design in Java* in a section called "Prefer a Tree to a Forest."

To illustrate, in Figure 3.8, if you do a simple copy of the object (called a *bitwise copy*), only the references are copied—not any of the actual objects. Thus, both objects (the original and the copy) will reference (point to) the same objects. To perform a complete copy, in which all reference objects are copied, you must write code to create all the subobjects.

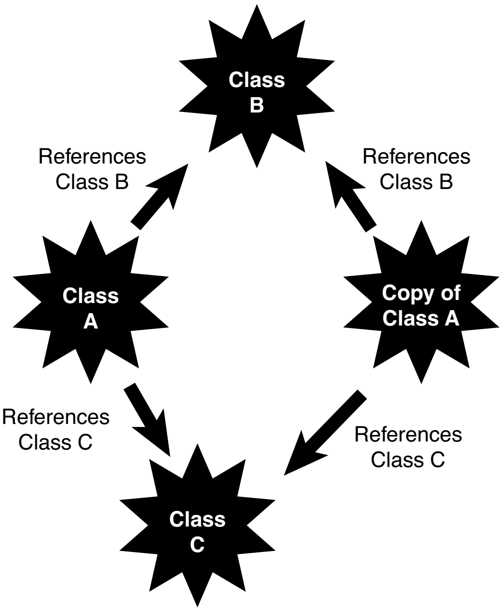

Figure 3.8 Following object references.

This problem also manifests itself when comparing objects. As with the copy function, this is not as simple as it might seem. Because objects contain references, these reference trees must be followed to do a valid comparison of objects. In most cases, languages provide a default mechanism to compare objects. As is usually the case, do not count on the default mechanism. When designing a class, you should consider providing a comparison function in your class that you know will behave as you want it to.

Conclusion

This chapter covered a number of advanced OO concepts that, although perhaps not vital to a general understanding of OO concepts, are quite necessary in higher-level OO tasks, such as designing a class. In Chapter 4, "The Anatomy of a Class," we start looking specifically at how to design and build a class.

References

- Meyers, Scott. 2005. *Effective C++*, 3rd edition. Boston, MA: Addison-Wesley Professional.
- Gilbert, Stephen, and Bill McCarty. 1998. *Object-Oriented Design in Java*. Berkeley, CA: The Waite Group Press.
- Tyma, Paul, Gabriel Torok, and Troy Downing. 1996. *Java Primer Plus*. Berkeley, CA: The Waite Group.

Example Code Used in This Chapter

The following code is presented in C# .NET. Code for other languages, such as VB .NET and Objective-C, are available electronically on the publisher's website. These examples correspond to the Java code that is listed inside the chapter itself.

The `TestNumber` Example: C# .NET

```
using System;

namespace TestNumber
{
    class Program
    {
        public static void Main()
        {

            Number number1 = new Number();
            Number number2 = new Number();
            Number number3 = new Number();
```

```
        }
    }

    public class Number
    {

        int count = 0;     // available to both method1 and method2

        public void method1()
        {
            count = 1;
        }

        public void method2()
        {
            count = 2;
        }

    }
}
```

This page intentionally left blank

Index

A

B

C

F

G

H

I

P

R

S

T

U

V-W

X