

Module 1 sample

Browsing with HttpClient

1.1 A taste of HttpClient: downloading a web page	1
1.2 Reviewing the HttpClient RFCs	3
1.3 The HttpClient API	14
1.4 HttpClient in action	19
1.5 Summary	40
Index	41

Ah! HTTP.

I can hear the sighs of relief and the shakes of head in an affirmative action, as you contemplate the title of this module. Yes, it's about HTTP—something, as developers, most of us are intimately aware of: the Hypertext Transfer Protocol that drives the thing we lovingly call the Internet. Specifically, it's about the Commons component HttpClient. When you think about it, this is the best possible name for the component; it exactly demonstrates what it does. *HTTP + Client* = HttpClient—a component to help deal with the client side of HTTP.

But wait a minute. I can hear you ask, “Isn't my browser an HTTP client?” Of course, it is. In fact, with the help of HttpClient, you can build your own browser, or embed it into your distributed application, or write a web services client. With the help of HttpClient, you can do anything that you want to do over HTTP (on the client side, anyway).

To start this module, we'll look at the simplest case of using HttpClient, by making it act as a browser and letting it download a web page. We'll then take a step back and take a brief look at the basics of HTTP: specifically, the Requests for Comments (RFCs) that HttpClient implements.¹ This will help you understand the fundamentals of HTTP and better prepare you for learning about HttpClient. We'll then take a whirlwind tour of the HttpClient API, to see its various packages and how they relate to each other. This will get you ready for the examples of using HttpClient that will follow.

1.1 A taste of HttpClient: downloading a web page

In this section, we'll look at how to use HttpClient in the simplest possible case of downloading a static web page. This is just a teaser section. More detailed examples will follow after we've discussed the HttpClient API.

Listing 1.1 shows the code required to download a web page using HttpClient. In this listing, HttpClient acts as a browser by requesting Google's home page.

Listing 1.1 A taste of HttpClient: Downloading Google's home page

```
package com.manning.common.chapter01;

import org.apache.commons.httpclient.HttpClient;
import org.apache.commons.httpclient.methods.GetMethod;
```

¹ See module 2, section 2.1.1, for a definition of an RFC.



```

public class HttpClientTeaser {

    public static void main(String args[]) throws Exception {
        HttpClient client = new HttpClient();
        GetMethod method = new GetMethod("http://www.google.com");
        int returnCode = client.executeMethod(method);
        System.err.println(method.getResponseBodyAsString());
        method.releaseConnection();
    }
}

```

HttpClient relies on three other Commons components for it to work. Before you run listing 1.1, make sure that you have the following libraries in your CLASSPATH, in addition to commons-httpclient.jar: commons-lang.jar (discussed in module 7, “Enhancing Java core libraries with collections”), commons-logging.jar, and commons-codec.jar (discussed in module 10, “Codec: encoders and decoders”).

If you run this Java application on a command line, the HTML for Google’s default page will be printed (provided you’re connected to the Internet). A browser would interpret the tags in this HTML and format its display accordingly, but here we were only interested in getting the HTML. We could just as easily have asked for a text document, image, or results of scripts—anything available on the Internet via HTTP—and HttpClient would have fetched it for us.

Using HttpClient starts with creating an instance of the client, as in the listing. In this case, the client that is created uses default settings. These settings include the connection manager used to create underlying connections, the attributes of these connections, the timeouts for giving up delayed connections, and so on.

Next, the GetMethod class is used to specify the document to retrieve using the instance of HttpClient that was just created. The GetMethod class implements the HTTP GET method for retrieving documents of the Internet. (Not surprisingly, you would use the PostMethod class for the HTTP POST method.)

The document to retrieve is specified as the sole argument to the GetMethod constructor. Notice that in this case, the argument specifies an absolute web address, as opposed to a relative web address, to Google’s default page (the argument must be interpreted as `http://www.google.com/index.html` or `http://www.google.com/index.htm` or whatever Google uses as its default page).

The HttpClient instance is used next to execute the method request to the default page of Google. HttpClient’s `executeMethod` expects an `HttpMethod` interface implementation as its argument and returns an integer, which represents an HTTP response code for the execution of this request. The GetMethod class extends the abstract class `HttpMethodBase`, which implements the `HttpMethod` interface.

Finally, the response from Google is printed on the command line by using the method `getResponseBodyAsString` on the GetMethod instance created earlier. This method evaluates the response received from the execution of a request and returns it by converting the bytes received into their `String` representation based on the character set specified in the response’s `Content-Type` header. You could choose to receive this response as a stream by using the `getResponseBodyAsStream` method, or as raw bytes by using the method `getResponseBody`.

As you can see, you need only four lines of code to retrieve a web page from the Internet using HttpClient. However, there is much more to HttpClient than simply retrieving web pages. Before we get into these extras, let’s take a brief look at the RFCs that HttpClient implements. These RFCs define HTTP, and you need to understand them before you can begin to understand HttpClient itself.

Module 2 sample

Uploading files with FileUpload

2.1 HTTP and uploading files	1
2.2 FileUpload basics	6
2.3 Uploading files without using FileUpload	10
2.4 Uploading files using FileUpload	13
2.5 Handling a complex input form	16
2.6 Uploading large files	20
2.7 Uploading files to a database	22
2.8 Summary	27
Index	29

The ability to transfer files from a user's PC to a remote server over the Internet allows the user to share images, documents, and audio-video files with other users around the world. The FileUpload component is an integral part of this transfer and sits on the server end. It receives, validates, and processes upload requests in a seamless fashion.

In this chapter, we'll review the basics of transferring files from a user's computer to a server, and you'll learn about rfc1867, the standard on which FileUpload is based. You'll then see an example of uploading files *without* using the FileUpload component. This will get you ready for the FileUpload component itself. We'll discuss this component with some basic examples and advanced usage scenarios.

2.1 HTTP and uploading files

File upload is the transfer of files from a user's computer to a server via an Internet browser. It's a subset of *file transfer*, which is the process by which files are transmitted between different machines. These machines must be interconnected by an underlying networking mechanism and must have software on either end that facilitates this transfer. A browser on a client machine, for example, is the high-level software that transfers files to a remote machine, which has server software running on it that handles the incoming files. To fully understand file transfer, it's important to have a working knowledge of the underlying protocol that file transfer works with.

Hypertext Transfer Protocol (HTTP) is this underlying protocol over which a browser transfers files. As opposed to HTTP, the File Transfer Protocol (FTP) is more specialized and geared toward the explicit transfer of files between machines. We'll discuss FTP further in module 3, "Handling protocols with the Net component," when we discuss the Commons Net component.

HTTP works on the basis of a request/response architecture. The user's browser sends a request message to a server for a particular resource, and the server responds with a response message to the original request. The format of these messages is standard. The browser sends the request to the server by encoding the user request in a simple fashion. This message contains the usual HTTP headers, and the user data is encoded using URL encoding. The server responds with a message that contains, in addition to the headers, the actual data in a Multipurpose Internet Mail Extension (MIME)-like format.

File transfer is inherently bidirectional and implicit. Think about what happens when you request a web page over the Internet through your browser. The browser makes a request to retrieve the requested page from



the server. The browser then makes a separate request for each element within that page and transfers each of these elements to your machine. By transferring files from the server to your machine, the browser is acting as a file transfer agent. Each image, HTML file, or data file is retrieved from the server, transferred over the Internet, and stored on your machine. Thus file transfer is happening behind the scenes even though you may not realize it. When you request that a file be transferred from the server to your machine, you're downloading files: This is *explicit* file transfer, as opposed to the *implicit* file transfer/download that the browser does on your behalf when you request a web page.

But how is file transfer inherently bidirectional? The underlying protocol doesn't put any restrictions on the order of transfer of files between machines. Files can be transferred in either direction and are transferred whenever a transaction takes place (it helps in this analogy to think about a file as blob of data). A browser can upload *and* download files; a server can receive *and* send files.

Let's look at how a simple file upload request is initiated and processed. A user requests a page that contains an HTML element of type `<input type="file">` and a form with an encoding of type `multipart/form-data` instead of the normal `application/x-www-form-urlencoded`. This page is rendered by the browser as part of a form element that allows the user to navigate his computer and select a file for transfer. The user then submits this form to the server. Now the browser takes over by converting this selected file into a MIME format for transfer to the server. At the server, this form is parsed, and the different parts of this form are processed. A Common Gateway Interface (CGI) program does this processing by interpreting the different parts according to their content type. The semantics of how file upload is handled are defined by rfc1867.

2.1.1 Understanding rfc1867

To understand how a browser handles a file-upload request, you need to first understand the rules that a browser follows to bundle the user's request to upload files so that files can be transferred over the Internet using the HTTP protocol. These rules are defined by rfc1867.

A Request for Comments (RFC) is an Internet Engineering Task Force (IETF) document that relates to creating standards for the Internet. RFCs define the many protocols, standards, and procedures that have become the guiding force of the Internet today. Protocols like HTTP (rfc1945), standards like MIME (rfc2045), and procedures like the Internet Standards Process (rfc2026) all began as RFCs published by the IETF. You can access these documents by logging on to www.faqs.org/rfcs.

Rfc1867 defines how a browser handles file upload requests from page authors and how a browser is supposed to retrieve and parse this information from the client's computer and send it to the server. It's an extension of the original HTML and MIME formats and is the standard by which the server parses the files sent to it.

This RFC enables file upload from HTML forms using two mechanisms. The first deals with what a page author needs to do to enable file upload and how such a request is represented by a browser to the end user. The second defines how a browser then formats the form for upload to the server. Both of these mechanisms are necessary for a fully functional and successful file upload. The program on the server that is responsible for parsing these formatted requests needs to be aware of the format and parse the request accordingly. The FileUpload component does this by adhering to this RFC.



Module 3 sample

Handling protocols with the Net component

3.1 Getting to know the protocols.....	1
3.2 The Net API	13
3.3 Creating a multiprotocol handler	23
3.4 Summary	32
Index	33

The Net component brings together implementations for a diverse range of Internet protocols. It's a feature-rich component and had been around a long time before it was open-sourced through Jakarta Commons. (It was originally built by ORO, Inc.)

Most programmers have to deal with only a subset of the vast array of Internet protocols. The ones that are used most often include Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP), and, perhaps, the mail protocols: Simple Mail Transfer Protocol (SMTP) and Post Office Protocol (POP3). However, several lesser-known protocols are also in use. The Net component features both the well known and the not so well known protocols in an easy-to-use interface.

In this chapter, we'll explore the Net component and look at the protocols it makes available. This will help you understand the basics of these protocols before you begin using them. Next, we'll examine the structure of the Net component and explore its API. Finally, we'll use all this information to develop a multiprotocol handler that uses each of the protocols.

3.1 Getting to know the protocols

A *protocol*, in the real world, is a way something should be done. By adhering to a protocol, you're following a strict procedure for doing certain things—for example, you shouldn't take the last helping of dessert without asking a dinner guest if they'd like to have it!

In the computer world, a protocol is a previously agreed upon way for two machines to exchange information with each other. Without a protocol to guide and define how machines talk to each other, there would be anarchy and confusion. A protocol may determine several things: how the two machines will handshake, how the transmitting machine will initiate transfer of data, what the format of the data will be, what the recipient machine will do to indicate that it has received the data, what the recipient machine will do to indicate an error condition, and so on.

3.1.1 TCP and UDP: the building blocks

Before we talk about the protocols covered by the Net component, let's discuss the protocols that are the building blocks of these protocols. A discussion of network technologies is incomplete without a basic understanding of the TCP and UDP protocols, which are the low-level protocols that act as the message carriers for the high-level, application-specific protocols.



The Transmission Control Protocol (TCP) is specified in rfc793.¹ It's a reliable, connection-oriented, complex protocol:

- It's *reliable* because data sent by TCP can't be lost: The protocol marks all packets of information that it transmits with a sequence number. This allows for retransmission of packets that go missing, because the receiving end can ask for those packets by looking up the sequence numbers of the packets it receives.
- It's *connection-oriented* because a connection must be established between machines before data can be exchanged between them.
- It's *complex* because it requires error correction and retransmission policies built in at the protocol layer itself.

The User Datagram Protocol (UDP) is specified in rfc768. It's a nonreliable, connectionless, simple protocol:

- It's *nonreliable* because packets sent via UDP may or may not reach their destination: The packets may get lost along the route due to incorrect addressing or checksum errors.
- It's *connectionless* because each packet is self-contained with the source host and destination host address. No direct connection stream is established between the source and the destination machines.
- It's *simple* because it doesn't require a connection, and no error checking or retransmission of packets is involved. This also means that if an application-level protocol is using UDP as the underlying protocol for transmission of data, it must be ready to accept loss of data or handle error checking and retransmission of lost data on its own.

It helps to think of TCP as a continuous phone line communication channel and UDP as a standard postal letter communication channel. With TCP, a continuous full duplex (two-way) channel is established before any data is exchanged, similar to numbers being dialed before a phone call begins. With UDP, a letter is marked with a destination address and is posted; it will probably reach its destination, but it may not.

Both TCP and UDP are protocols on the Transport layer of the standard TCP/IP four-layer model, which is shown in figure 3.1. For this reason, these protocols are low-level protocols as compared to, for example, HTTP, which is considered a high level protocol.

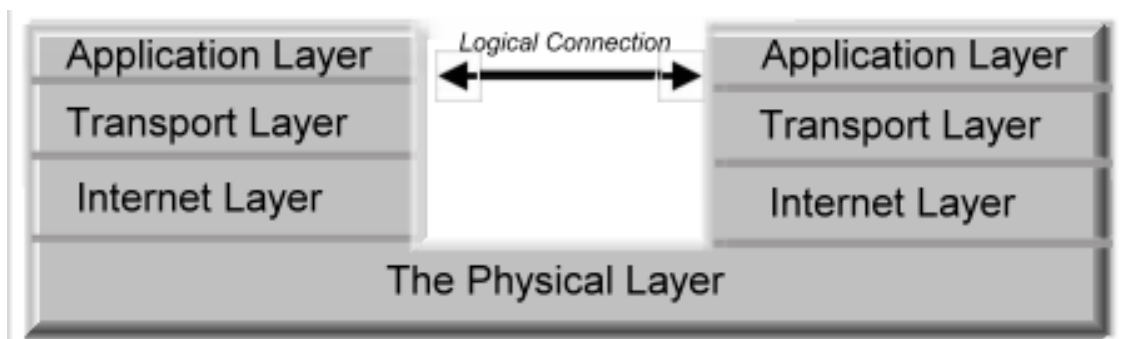


Figure 3.1 The four-layer TCP/IP model, showing the different levels of protocols

3.2.2 High- and low-level protocols

Whenever an application-level protocol is used, it's likely to be running on top of other low-level protocols. Consider HTTP. Although this protocol is designed to transfer information between a web browser and a

¹ See module 2, section 2.1.1, for the definition of an RFC.

Module 4 sample

XML parsing with Digester

4.1 The Digester component	1
4.2 The Digester stack	5
4.3 A match made in heaven?	6
4.4 Conforming to the rules!	7
4.5 Resolving conflicts with namespace-aware parsing	22
4.6 Rule sets: sharing your rules	23
4.7 Externalizing patterns and rules: XMLRules	24
4.8 Summary	26
Index	27

Configuration files are used in all sorts of applications, allowing the application user to modify the details of an application or the way an application starts or behaves. Since storing these configuration details in the runtime code is neither possible nor desirable, these details are stored in external files. These files take a variety of shapes and forms. Some, like the simple name-value pair, let you specify the name of a property followed by the current assigned value. Others, like those based on XML, let you use XML to create hierarchies of configuration data using elements, attributes, and body data.

For application developers, parsing configuration files and modifying the behavior of the application isn't an easy task. Although the details of the configuration file are known in advance (after all, the developers created the basic structure of these files), objects specified within these configuration files may need to be created, set, modified, or deleted. Doing this at runtime is arduous.

The Digester component from Jakarta Commons is used to read XML files and *act* on the information contained within it using a set of predefined rules. Note that we say XML files, *not* XML configuration files. Digester can be used on all sorts of XML files, configuration or not.

The Digester component came about because of the need to parse the Struts configuration file. Like so many of the Jakarta Commons projects, its usability in Struts development made it a clear winner for the parsing of other configuration files as well.

In this module, we'll look at the Digester component. We'll start with the basics of Digester by looking at a simple example. We'll then look at the Digester stack and help you understand how it performs pattern matching. This will allow us to tackle all the rules that are prebuilt into Digester and demonstrate how they're useful. We'll use this knowledge to create a rule of our own. We'll round out the module by looking at how the Digester component can be externalized and made namespace-aware.

4.1 The Digester component

As we said, the Digester component came about because of the need to parse the Struts Configuration file. The code base for the parsing the Struts configuration file was dependent on several other parts of Struts. Realizing the importance of making this code base independent of Struts led to the creation of the Digester component. Struts was modified to reuse the Digester component so as not to include any dependencies.

In essence, Digester is an XML → Java object-mapping package. However, it's much more than that. Unlike other similar packages, it's highly configurable and extensible. It's event-driven in the sense that it lets



you process objects based on events within the XML configuration file. *Events* are equivalent to patterns in the Digester world. Unlike the Simple API for XML (SAX), which is also event-driven, Digester provides a high-level view of the events. This frees you from having to understand SAX events and, instead, lets you concentrate on the processing to be done.

Let's start our exploration of the Digester component with a simple example. Listing 4.1 shows an XML file that we'll use to parse and create objects. This XML file contains bean information for the JavaBeans listed in listings 4.2 and 4.3. Listing 4.4 shows the Digester code required to parse this file and create the JavaBean objects.

Note: To be able to run these and all the other examples in this module, you need to have two supporting libraries in addition to commons-digester in your CLASSPATH. These libraries are common-logging.jar and commons-collections.jar.

Listing 4.1 A simple XML file (book_4.1.xml)

```
<?xml version="1.0"?>
<book title="Jakarta Commons in Action">
  <author id="1001">
    <name>Vikram Goyal</name>
  </author>
</book>
```

Listing 4.2 Book JavaBean

```
package com.manning.common.chapter04;

import java.util.Vector;

public class Book {

    private String title;
    private Vector authors; ← List of book's authors held in this Vector

    public String getTitle() { return this.title; }
    public void setTitle(String title) { this.title = title; }

    public Vector getAuthors() { return this.authors; }
    public void setAuthors(Vector authors) { this.authors = authors; }

    public void addAuthor(Author author) {
        if(authors == null) authors = new Vector();
        authors.add(author);
    }

    public String toString() { ← Book bean returns toString representation
        StringBuffer buffer =           that includes title and all authors
        new StringBuffer("Title: " + getTitle() + "\r\n");

        if(authors != null) {
            for(int i = 0; i < authors.size(); i++) {
                buffer.append(
                    "Author " + (i + 1) + ": " + authors.get(i));
            }
        }
    }
```


Module 5 sample

JXPath and Betwixt: working with XML

5.1 The JXPath story	1
5.2 The Betwixt story	21
5.3 Summary	34
Index	35

The need to cover two components in this module originated from the consolidation of similar components. JXPath and Betwixt, although different in concept, are complementary technologies that enable a higher degree of interaction between Java objects and XML data. The two technologies, used together, go a long way in simplifying the Java developer's life.

Betwixt resembles Digester, which we covered in the previous module, by providing a similar functionality of converting XML data to Java objects. JXPath is a handy tool if you want shorthand syntax for complex object access.

JXPath and Betwixt solve small but very important pieces of a puzzle. Betwixt helps convert XML data to and from Java objects, whereas JXPath provides an expression language for making sense of complex data structures and objects. Betwixt can convert your data into a readable format, and JXPath can help by accessing this data—even the most complex parts of it.

We'll start with JXPath. It's based on XPath, which is the syntax for traversing XML documents. Therefore, we'll cover the basics of XPath before we dive into JXPath.

We'll explore Betwixt from two angles: the XML-to-Java path and the Java-to-XML path. We'll help you understand how Betwixt operates by first explaining the concept of data binding and then diving into its structure. All through this discussion are plenty of examples to demonstrate how Betwixt operates.

5.1 The JXPath story

JXPath, like the other Commons components, solves a very focused problem for the Java developer. It lets you simplify access to complex nested objects through a well-defined architecture. This architecture derives heavily from XPath, which we'll introduce in section 5.1.2.

Let's start our introduction of JXPath by looking at a real-world problem that we'll use as the basis for discussing this Commons component. As you understand more about JXPath, we'll enhance the application to handle more complex functionality.

5.1.1 Understanding the problem: the Meal Plan application

The whole idea of JXPath is to easily define paths to complex objects. Keeping this in mind, let's create an application that tracks objects, which are hard to access unless you use complex iterations and loops.

The goal is to create a Meal Plan application. The application specifies the meals to be eaten by a person on a diet and tracks the ingredients of each meal. The `MealPlan` class has start and end dates that default to



the current date and seven days from the current date, respectively. You can add meals to the meal plan. Each `Meal` class has an attribute for the day that meal is to be eaten (Sunday = 0, Saturday = 6), the type of meal it is (breakfast = 0, lunch = 1, dinner = 2, other = 3), and the name of the meal. Each meal can contain several ingredients. A key, a name, and an alternate qualify each `Ingredient` class. Figure 5.1 shows a simplistic relationship diagram for these domain objects.

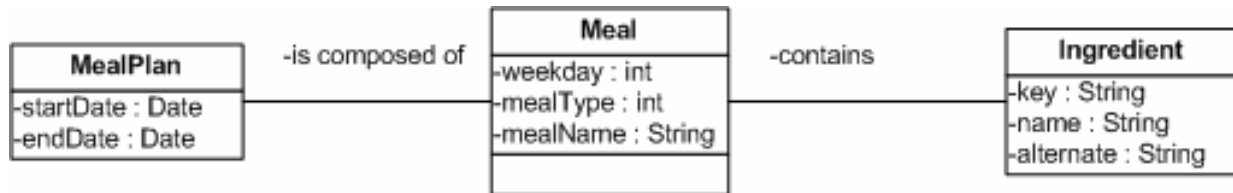


Figure 5.1 Class diagram for the Meal Plan application problem that shows the relationships between domain objects

Let's look at the full source code for these classes. This is our first attempt at creating these objects, and we'll refine them as we explain more about the application and XPath. We'll start with the `MealPlan` class, shown in listing 5.1.

Listing 5.1 The `MealPlan` class

```

package com.manning.common.chapter05;

import java.util.Map;
import java.util.Date;
import java.util.Iterator;
import java.util.Calendar;
import java.util.ArrayList;
import java.util.Collection;
import java.util.GregorianCalendar;

public class MealPlan {
    private Date startDate;
    private Date endDate;
    private Collection meals;    ❶ Declare meals collection

    public Date getStartDate() {
        return this.startDate;
    }
    public void setStartDate(Date startDate) {
        this.startDate = startDate;
    }
    public Date getEndDate() {
        return this.endDate;
    }
    public void setEndDate(Date endDate) {
        this.endDate = endDate;
    }
    public Collection getMeals() {
        return this.meals;
    }
    public void addMeal(Meal meal) {
        if(meal == null)
            throw new IllegalArgumentException("Meal cannot be added null");
    }
}
  
```

Module 6 sample

Validating data with Validator

6.1 The data validation process.....	1
6.2 The Validator component	3
6.3 Validator in action	6
6.4 Summary	25
Index	26

In the world of programming, once in a while a nifty idea comes along—a valuable tool or a spark of imagination that makes you wonder how you ever programmed without it. Commons Validator is one such tool; once you start using it, it’s difficult to program without it.

Validator provides a guideline and API for validating user data. It forces the developer to centralize and streamline data validation and lays down the best possible way to manage this validation. At the same time, it provides an API that brings order to data validation. It really is the tool that you can’t live without, if your application regularly accepts user data.

In this module, we’ll start with a little background about data validation—why you should do it, how to do it, and what should be done. We’ll then move on to discuss the Validator component, giving an overview and discussing its API. This will prepare you for the Validator examples that follow. You’ll see how to use Validator in several situations, including web-based and normal applications. In the web applications, we’ll consider Struts and non-Struts applications, to cover Validator usage in almost all situations you’re likely to encounter.

6.1 The data validation process

Computers are dumb. Yes, that’s right. Computers are dumb machines that do what you and I tell them to do. So why is it so difficult for us to make them work the way we want them to? Is the problem something we do, or something we don’t do? The answer isn’t easy to figure out. How many of us have spent hours of frustrating detective work trying to figure out what the computer is doing, only to realize that it was after all, a stupid mistake on our part that made it behave that way? Yes, it’s not computers that are dumb: It’s us.

With that frightening realization out of the way, let’s see how we can minimize the time and effort required to figure out problems in our code. Since every application depends on the data it receives, the first salvo to fire is targeted toward making sure this data is validated to conform to set guidelines. This is why data validation processes are the most important facets of any application.

Let’s start our discussion of the process with an understanding of why data validation should be performed.

6.1.2 Why bother?

In simple terms, data should be validated because users are unpredictable. You can’t rely on users to consistently provide you with valid data that’s applicable for your application. Even the most fastidious users will tire and input invalid numbers in a salary field, for example. However, data should be validated for a number of reasons to cover a variety of issues that you’re likely to encounter:



- To cater to unpredictable users. For example, a user may enter a `String` value in a `Number`-only field, due either to ignorance or a genuine mistake.
- To constantly protect your application from rogue and mischievous users.
- To ensure the integrity of data already in your system. There is nothing more frustrating than sifting through your application's data store for invalid data that has the potential to render your data systems invalid.
- To ensure conformity and compliance with existing systems.
- To reduce application-processing time. In most systems, frameworks exist to separate the task of business processing and front-end interactions into separate modules. If data validation at the front end can sift through the user input and invalidate entries that don't conform to set guidelines, it reduces the processing that business units have to do.

This is by no means an exhaustive list of the reasons for data validation, but it covers the most prominent ones. In the next section, we'll discuss how to perform data validation.

6.1.2 The how-to of data validation

We all realize the need to perform data validation and agree that's a required part of the application development process. But how do we perform validation? Following are some basic guidelines for this process:

- The first step in data validation should be performed at the data-entry point. There is no sense in holding data and performing validation at a later stage, because any subsequent processing can be avoided if the supplied data fails. The only exception to this rule arises when the validity of data depends on subsequent data collection or business process.
- You should take into account existing criteria for data-set validation. This means that validation can only be performed against rules governing a collection of previous data. These rules must be explicit and precise and should be made available to the validation routines before data is gathered from the user.
- The data validation rules must also be separated from the source code so you can modify the validation rules without affecting applications that are in production mode. Therefore, these rules must be supplied to the validation routines as an external process that's only looked up at the precise moment the validation is to be performed.
- The data validation process should also make sure the user or system supplying the data can be made aware if any data validation checks and consequently can be allowed to resubmit with correct data or gracefully withdraw from continuing with the application.
- The data validation routines must be independent of the surrounding process. This means that the same validation routines should work identically in all applications, regardless of the way input is supplied to the routines.

Let's now discuss what to look for when you're validating data.



Module 7 sample

Enhancing Java core libraries with Collections

7.1 A taste of things to come	1
7.2 Collections and more	3
7.3 Summary	31
Index	32

The Commons components we'll discuss in this module and the next have one thing in common: They cover the deficiencies of the core Java classes by building wrappers around them or extending them to provide sets of useful libraries. These components are BeanUtils, which provides wrappers around the Java Reflection and Introspection API; Collections, which enhances the Java Collection API; and Lang, which provides helper utilities for manipulation of the core Lang package.

In this module, we'll study the Collections component; we'll leave the discussion of the other two components to the next module. All three of these components complement the Java core API. These are very hands-on modules and, in that sense, are different from the rest of this book, because we don't need to cover any background material before showcasing the examples. We'll start with a simple example that uses part of each component to break the ice, and then we'll move to the Collections component.

7.1 A taste of things to come

As we said before, this module and the next differ from the other modules in this book because they deal with components that aren't application-oriented. This means the components of these two modules don't lend themselves to complete application development like the Modeler (module 11) or XPath (module 5) Commons components. The Collections component classes and API can be used as utilities and helper routines to build applications and extend the core Java API in everyday functions.

We'll start with a small example that brings all of these components together, as shown in listing 7.1.

Listing 7.1 Using the Collections, BeanUtils, and Lang components together

```
package com.manning.common.chapter07;

import java.util.Map;
import java.util.HashMap;
import java.util.Iterator;
import java.lang.reflect.Method;

import org.apache.commons.collections.Bag;
import org.apache.commons.collections.bag.HashBag;

import org.apache.commons.beanutils.BeanUtils;
import org.apache.commons.beanutils.PropertyUtils;
```



```

import org.apache.commons.lang.StringUtils;

public class TasteOfThingsV1 {

    private static Map testMap;
    private static TestBean testBean;

    public static void main(String args[]) throws Exception {
        prepareData();

        HashBag myBag = new HashBag(testMap.values());

        System.err.println("How many Boxes? " + myBag.getCount("Boxes"));
        myBag.add("Boxes", 5);
        System.err.println("How many Boxes now? " + myBag.getCount("Boxes"));

        Method method =
            testBean.getClass().getDeclaredMethod("getTestMap", new Class[0]);
        HashMap reflectionMap =
            (HashMap)method.invoke(testBean, new Object[0]);
        System.err.println("The value of the 'squ' key using reflection: "
            + reflectionMap.get("squ"));

        String squ = BeanUtils.getMappedProperty(testBean,
            "testMap", "squ");
        squ = StringUtils.capitalize(squ);
        PropertyUtils.setMappedProperty(testBean,
            "testMap", "squ", squ);
        squ = BeanUtils.getMappedProperty(testBean,
            "testMap", "squ");
        System.err.println("The value of the 'squ' key is: " +
            BeanUtils.getMappedProperty(testBean, "testMap", "squ"));
        String box = (String)testMap.get("box");
        String caps =
            Character.toUpperCase(box.charAt(0)) +
            box.substring(1, box.length());
        System.err.println("Capitalizing boxes by Java: " + caps);

        private static void prepareData() {
            testMap = new HashMap();
            testMap.put("box", "boxes");
            testMap.put("squ", "squares");
            testMap.put("rect", "rectangles");
            testMap.put("cir", "circles");

            testBean = new TestBean();
            testBean.setTestMap(testMap);
        }
    }
}

```

Create TestBean with a TestMap as a property

← Create Bag that counts its contents

① Bag working

② Get map value with Java reflection

③ Combine BeanUtils and Lang to change map value

④ Show capitalization with Java

The code listing not only mixes the three components together, but also shows how equivalent actions would be done in native Java code if these components weren't available. The code creates a TestBean JavaBean

Module 8 sample

Enhancing Java core libraries with BeanUtils and Lang

8.1 BeanUtils: JavaBeans made easy.....	1
8.2 Mind your Lang(uage)	22
8.3 Summary	36
Index	37

This module continues where the previous one ended, by continuing to discuss the Commons components that enhance the Java core libraries. In the previous module, we discussed the Collections component. In this module, we'll look at the BeanUtils component, which is used to supplement the Java Reflection and Introspection API; and Lang, which provides helper classes for the Java Lang package.

Let's start with BeanUtils. We'll take a quick tour of its API and packaging and then delve into some examples.

8.1 BeanUtils: JavaBeans made easy

Although BeanUtils is really a supplement to the Java Reflection and Introspection API, its main functionality is directed toward the ability to read and write JavaBeans without needing to know the name and signature of their properties or methods.

BeanUtils (short for Bean Introspection Utilities) relies on the JavaBeans that it works on to follow the standard JavaBean specification. The specification puts some restrictions on the structure of JavaBeans. However, BeanUtils is lenient in terms of these restrictions and only requires the following two to be enforced:

- A JavaBean must have a no-argument constructor, and its class must be public.
- The properties of a JavaBean must follow a standard pattern. This pattern ensures that a property can be accessed by using the method `getXXX` (or `isXXX` for boolean properties), where `XXX` is the property name. Similarly, properties can be modified by using the method `setXXX`. These methods are typically called the accessor and mutator methods, respectively. Note that there must be only one accessor or mutator for each property; overloaded methods will fail to work with BeanUtils.

The current version of BeanUtils is 1.7.0. With this release, BeanUtils has been separated into two libraries. The first library, `commons-beanutils-core.jar`, contains the core BeanUtils classes. The second library, `commons-beanutils-bean-collections.jar`, is an add-on that allows you to use BeanUtils with the Collections component. If you don't plan to use BeanUtils with the Collections API, it's recommended that you use the core library, because it removes the dependency on the Collections component. In this module we'll use both libraries, using the combined jar file `commons-beanutils.jar`. This means that the Collections components jar file must also be in your CLASSPATH in order for any of the Collection-BeanUtils examples to work.



Before we look at any examples, you must first understand the terminology associated with JavaBeans and BeanUtils.

8.1.1 Understanding the terminology

Simple. Scalar. Indexed. Nested. Mapped. DynaBean. *Lazy* DynaBean. If these terms leave you confused in relation to JavaBeans and BeanUtils, then this section will give you a quick overview of what they all mean. It's important that you have a clear understanding of these terms, because we'll use them a lot later in this module. Therefore, with the help of an example, we'll explain them here. If you already understand these terms, then it's safe to skip this section, keeping in mind that this section also introduces the JavaBeans we'll work with while explaining the BeanUtils examples.

Property types

JavaBean properties can be divided into four types, based on what they represent. These four types are scalar (or simple), indexed, mapped, and nested. Before we describe these types, look at listings 8.1, 8.2, and 8.3, which show three example JavaBeans.

Listing 8.1 Going to the movies with the Movie JavaBean

```
package com.manning.common.chapter08;

import java.util.Map;
import java.util.List;
import java.util.Date;

public class Movie {
    public Movie() {
    }

    public Date getDateOfRelease() { return this.dateOfRelease; }
    public void setDateOfRelease(Date dateOfRelease) {
        this.dateOfRelease = dateOfRelease;
    }

    public String getTitle() { return this.title; }
    public void setTitle(String title) {this.title = title; }

    public Person getDirector() { return this.director; }
    public void setDirector(Person director) { this.director = director; }

    public List getActors() { return this.actors; }
    public void setActors(List actors) { this.actors= actors; }

    public String[] getKeywords() { return this.keywords; }
    public void setKeyWords(String[] keywords) { this.keywords = keywords; }

    public Map getGenre() { return this.genre; }
    public void setGenre(Map genre) { this.genre = genre; }

    private Date dateOfRelease;
    private String title;
    private Person director;
}
```

Module 9 sample

Pool and DBCP: creating and using object pools

9.1 The Pool component	1
9.2 The DBCP component	18
9.3 Summary	37
Index	39

Pool and DBCP are the Jakarta Commons components that deal with creating, managing, and using object pools. DBCP (which stands for Database Connection Pool and uses the Pool component to manage database connections) represents a successful implementation of the Pool component because it builds on it to supply a connection-pooling mechanism for managing database connectivity. The generalized nature of the Pool component is brought into narrow focus by DBCP. It's therefore natural for us to discuss these two components together.

The Pool component represents a generic API for implementing pools of objects. These objects can represent any Java class, and the Pool API provides the infrastructure to maintain these objects in a pool for easy access, easy return, and lifecycle maintenance.

The DBCP component also represents a generic API for maintaining connections to a database. As we said, it uses the Pool component to maintain these connections, but it also provides support services for maintaining interactions between a database and the application that is accessing it. It does so in a seamless fashion without needing to know the underlying database structure.

In this module, we'll start with the Pool component. We'll discuss object pooling and look at the structure of the Pool API, and we'll use it to create a basic employee pooling mechanism. We'll then move on to explore DBCP and build some concrete examples.

9.1 The Pool component

The Pool component contains a set of APIs that let you manage objects in a pool for reuse and maintainability. The idea of using a pool is to provide reusable objects. A pooling mechanism should not make any assumptions about the objects it contains and should provide facilities for easy creation, management, reuse, and destruction of those objects. The Pool component of Jakarta Commons is such a mechanism; it goes a step further by providing a means to create a factory for creation of the pool itself. But we're getting ahead of ourselves. Let's first look at the need for object pooling and consider some of the issues involved. We'll then examine the structure of the Pool API and discuss in detail the facilities provided by the Pool component.

9.1.1 The case for object pooling

Object pools provide a better way to manage available resources. You've probably come across object pools before in one form or another, most notably as connection pools for managing access to databases. However,



most developers use other object pools without realizing they're doing so: Application servers, servlet engines, mail servers, even database servers, all use some sort of thread-pooling mechanism to service the requests they receive. *Thread pooling* is a way by which most high-end servers manage to respond to requests for services quickly. These servers don't create a thread for each request that comes in; instead, they maintain a pool of ready threads that can be rapidly assigned to a request. The thread does its business and then is returned to the pool for future requests.

Object pooling relies on three development requirements as strong cases for its use:

- *Responsiveness*—Because objects are already created and in a pool, there's no overhead associated with object creation. Your application can quickly dip into the pool, pick an object, and assign it to the task at hand. Contrast this with the tasks needed if object pooling isn't used: Objects must be created and initialized, and only then can they be assigned to the task at hand. Complex objects are the worst offenders and will cause your application to suffer from poor response times. As with most things, there is a tradeoff in this scenario. Heavy objects are a drain on system resources, and maintaining them in system memory waiting for tasks can be problematic. You need to maintain a fine balance between the minimum objects required in the pool and your system's responsiveness.
- *Reusability*—Objects in a pool are reused. Although this may seem similar to the first feature, there is difference because of the way object pools are managed. Consider the case of an object pool where each time you took out an object, a new object was created to replace it (behind the scenes, by a separate thread). The object you took out wasn't returned to the pool but was discarded. Although this process would serve the first purpose of a responsive object pool, it would create overhead for the system. To avoid the overhead of object creation and subsequent garbage collection for the discarded objects, objects pools implement a mechanism by which the object borrowed from the pool can be returned. This returned object can then be used by other tasks.
- *Controlled access*—Objects in a pool are controlled because they're accessed and maintained within the pool. While the pool is sitting idle, it may perform maintenance operations like repairing broken objects, releasing unusable objects, and so on. Access to these objects is limited and controlled via the pool. A request for an object from the pool is handled internally using any protocol, without the application knowing anything about it. Further, by restricting access, you limit the number of these objects in the system at any time, thus improving performance and security. An object pool is an example of the Factory Pattern for creation of objects.

An object pool may not be the best way to manage your objects, however. Object pools are best used in cases where the objects that are being pooled are complex; have large initialization, creation, and destruction times; are used frequently for short small tasks; and can be recycled. Examples of some generic objects that should be pooled include database connections, socket connections, and threads in application servers. In the next section, we'll examine the difference between using an object pool to manage objects and letting the garbage collector in your JVM do its object-collection trick.

Object pooling vs. the garbage collector

The alternate to using an object pool is to let the garbage collector in Java do its business. You use the new keyword to create your objects, and the garbage collector cleans them up when they aren't needed. The humble garbage collector has seen many performance advances since its inception, to the stage that it's now a rival to object pools. A fast and efficient garbage collector can quickly reclaim objects that are no longer needed by your application, thus freeing up memory and making that memory available to other objects. This

Module 10 sample

Codec: encoders and decoders

10.1 What are encoding and decoding?	1
10.2 Understanding the Codec algorithms.....	2
10.3 Getting to know Codec	9
10.4 Codec at work	14
10.5 Summary	18
Index	19

Jakarta Codec is an effort to provide implementations for various encoders and decoders that a developer is likely to encounter. At the time of this writing, encoders and decoders are provided for Base64, Hex, phonetic encodings, and URLs. It's possible to extend this API to provide your own implementations for either the supported encodings or any other new encoding scheme.

In this module, we'll look at how to use the Codec component for various encoding/decoding scenarios. We'll start with the basics of each supplied encoding mechanism to help you understand the underlying technology; and we'll show an example of how to implement it using Codec.

10.1 What are encoding and decoding?

If you examine the headers of an email message, a typical header you're likely to encounter is Content-Transfer-Encoding. What is this header used for, and who puts it there? The header is added by the mail client you use to compose the message (in other words, the mail client of the message sender). It's added so that when the mail message passes through the various Internet mail transport mechanisms, it can be identified as having been encoded. This *encoded* message can then pass through these systems, which may have limitations on the data or character sets that they can handle. For example, binary data can't be transferred through these systems if it isn't encoded in a certain way (Base64). This header also identifies to the message recipient the *decoding* to be applied to re-create the original message. This is the basic crux of encoding/decoding. A system encodes a piece of information for safe transmission, conciseness, or security. The receiving system decodes that information based on the encoding used and re-creates the original information. In this section, we'll elaborate on these concepts.

Digital transmission of information requires textual data to be encoded so that differing systems can agree on and understand the information being transmitted. Encoding, however, isn't just required for textual data. Systems require information to be encoded in order to make it concise or unambiguous. Phonetic encoding of related words is an example of such encoding.

The basic premise of encoding is to convert information from one form to another. As we stated earlier, this second form may be relevant for either transmitting this data over channels that are more conducive to the converted form, or it may represent a more concise representation of the original data.

Information that is encoded must be marked with the rules used to do the encoding so that another system can use this marking to understand the encoding rules applied. This other system, can then apply these rules in reverse to arrive at the original information—in effect, decoding the encoded information.



The most prevalent form of digital encoding of textual data, and the one that you may have come across, is the American Standard Code for Information Interchange (ASCII). This encoding method was developed to represent English characters as numbers, with each letter assigned a number from 0 to 127. For example, the ASCII code for uppercase *A* is 65. With 26 letters in the English alphabet and room for more characters like the space and special symbols (such as \$ and %), ASCII is well suited for representing and transmitting textual English information. ASCII also works well to represent Latin-based languages. However, it falls short for other languages, like Arabic, Chinese, and Hindi, which are based on separate scripts and have hundreds of characters that can't be represented by the encoding scheme used in ASCII.

The Unicode encoding scheme (<http://www.unicode.org>) overcomes this problem. It defines three encoding forms that allow the same data to be transmitted in 8-bit, 16-bit, or 32-bit formats. With a theoretical limit of $2^{32} - 1$, Unicode is well suited to represent all the languages of the world.

10.2 Understanding the Codec algorithms

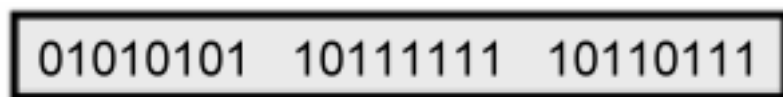
Before we delve into the Codec component, let's look at the algorithms supported by Codec. In the next few sections we'll look at the basics of each encoding scheme supported by Codec.

10.2.1 Base64

An email message that you type and send to a distant recipient is transported across the world through a series of Internet mail gateways that route your message based on header information. This information needs to be in a format that these gateways understand and can interpret for routing forward. However, considering the vast number of gateways around the world and the languages they work in, it's possible that messages would be lost if they weren't sent with headers that contain information in a standard way. It isn't just the headers that need to be in a standard language, though; transporting images and multimedia files requires the conversion of these binary files into transport-safe versions. Toward this end, the Base64 encoding mechanism is defined in rfc2045 (<http://www.ietf.org/rfc/rfc2045.txt>; see module 2 for the definition of an RFC). This RFC deals with the large issues of mail transport and provides other useful information as well. Base64 is only one of the recommendations that is included as part of the Content-Transfer-Encoding header mandate; the other encoding is quoted-printable.

Base64 allows you to convert binary (8-bit) data into a 7-bit short line format. This is necessary because mail transport disallows any other format. For example, you can't transfer mail messages that are binary in format or are represented using 8-bit encoding. This would typically happen if you were transporting media files or images, which, in their natural format, are either binary or 8-bit. Also, by encoding the images and other multimedia files with Base64, you get a textual representation of these files, which may be useful when you want to store the images in an archive.

Base64 works by dividing three continuous octets in the input data into sets of 6 bits each and then representing each 6 bits with the ASCII equivalent. This is better explained by an example. Consider the binary input for some arbitrary data shown in figure 10.1.



01010101 10111111 10110111

Figure 10.1 Arbitrary binary input

Our task is to convert this binary data into its Base64 equivalent. We start by splitting these 24 bits (3 octets = 24 bits) into groups of 6 bits each to arrive at 4 groups, as shown in figure 10.2.



Module 11 sample

Managing components with Modeler

11.1 Understanding JMX	1
11.2 Say hello to Modeler	20
11.3 Summary	31
Index	32

Imagine that you've just built the next killer Java application, and it's selling like hotcakes. However, even though (being a good developer) you built in enough debug and tracing information to monitor the state of your live application units, you're missing the ability to monitor each and every class and their attributes. Although you could build monitoring services as part of your application, it takes the focus away from your core business, the application itself. It would be nice if you could monitor your application for the state of its classes, attributes, and operations. Java Management Extensions (JMX) let you do just that. However, this chapter isn't about JMX. It's about how to easily configure JMX using the Modeler component. Specifically, it's about how to use the Modeler component to create Model MBeans (a special kind of management bean), which are used to monitor resources in your application.

It would be difficult to understand Modeler without first JMX. Therefore, this chapter starts with a simple tutorial of the JMX API and its components. We'll pay special attention to Model MBeans and how they interact with their environment. We'll then introduce Modeler using several examples. We'll round out the chapter with examples of how to integrate Modeler with the Ant build tool.

11.1 Understanding JMX

JMX is a toolset that provides guidelines to manage and monitor applications in a distributed environment. But what does that mean? It means that using JMX, you can view the state of your application, make changes to variables, invoke operations on your objects, and gather any other information that is exposed by your application, without making wholesale changes to your application's main functions or architecture.

Let's try to understand this in the context of an application with diagrams and code. In this process, we'll introduce the various layers of a JMX-managed application. The first layer is called the Instrumentation layer.

11.1.1 Introducing the Instrumentation layer

To begin, consider figure 11.1. It shows an application as a standalone component that's deployed and functional in a production environment.

How do you introduce JMX in this picture? You start by introducing another component called a managed bean (MBean). An MBean is a Java object that acts as a wrapper for your application components and exposes them for management. Figure 11.2 shows an MBean introduced into figure 11.1.



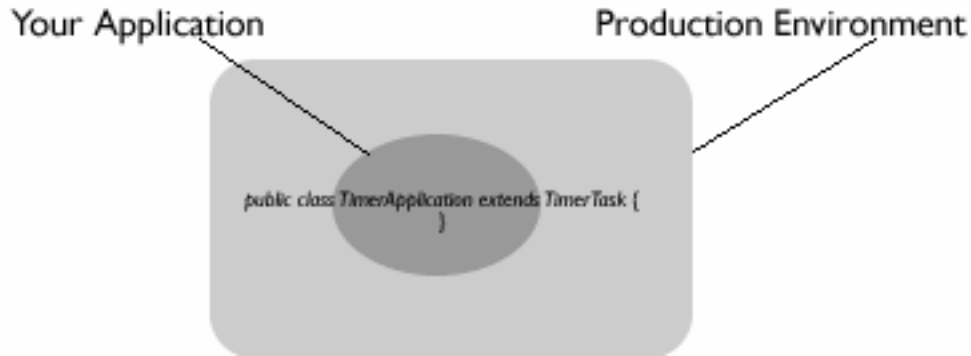


Figure 11.1 An application in a production environment. How do you manage this application?

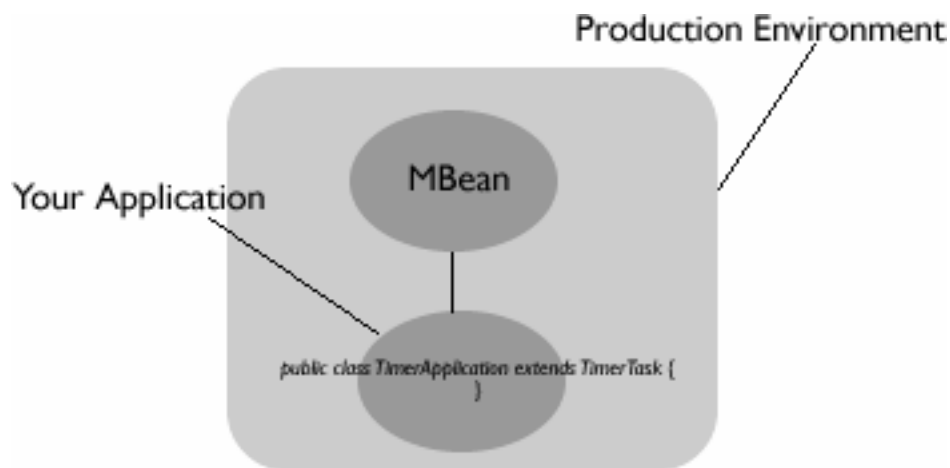


Figure 11.2 Introducing an MBean to expose your application for management

The MBeans and your application's components together form what is called the *Instrumentation layer* for JMX-managed applications. The Instrumentation layer interfaces with the outside world to expose these components for management. But how do the MBeans in this layer expose these components? Well, MBeans are interface descriptions of your components. This means that the MBeans closely resemble the components they expose and that the components must implement these interfaces themselves. These interface definitions are read by the outside components using Java Reflection.

Let's look at some code to clarify things before proceeding further. Listing 11.1 shows a simple application that prints the current date and time after a specified delay. Consider this listing an application component that will be managed using JMX.

Listing 11.1 Simple application that prints the current date and time after a delay

```
package com.manning.common.chapter11;

import java.util.Date;
import java.util.Timer;
import java.util.TimerTask;

public class TimerApplication extends TimerTask {

    private long delay;
```


Module 12 sample

Command-line processing with CLI

12.1 The process of command-line interface.....	1
12.2 Introducing CLI	5
12.3 Summary	25
Index	26

Applications don't work in isolation. A simple enough statement, which means that any application interacts with its environment, affecting the way it behaves. This interface to an application lets you modify application parameters that affect the application's behavior, either at start-time or at runtime. In this module, we'll look at the Command Line Interface (CLI) component from the Commons stable, which enables optimal processing of application parameters at an application's start-time. (The previous module on Modeler discussed how you can manage application parameters at runtime.)

We'll first look at command-line processing. It might seem that the process is simple, but in reality it involves three stages, as used by CLI. We'll then take a brief look at the CLI API. This will set us up for the CLI examples that follow.

12.1 Command-line processing

Command-line processing is the means by which an application understands its initial environment. It sets the stage for subsequent processing. The process that creates the application supplies the parameters—or, as they're known in CLI, the *options*—that define this environment. The application then parses these options and executes accordingly.

As defined by CLI, there are three stages in the processing of options on a command line for an application: definition, parsing, and interrogation. Figure 12.1 shows these three stages.

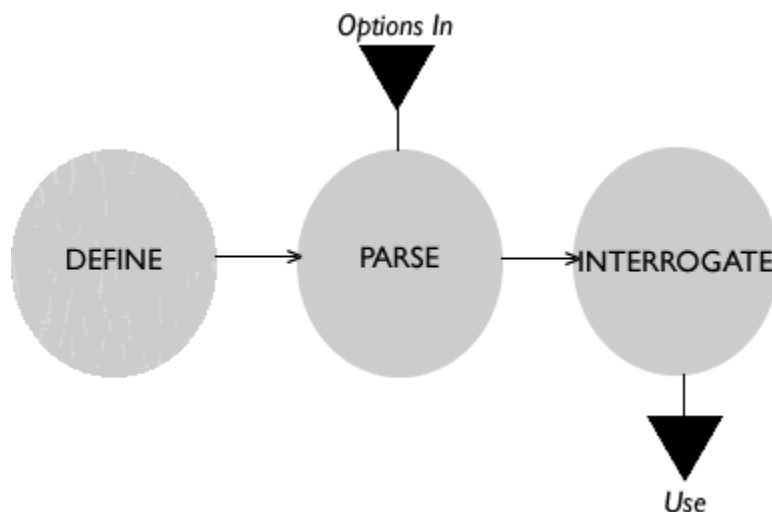


Figure 12.1 The three stages of command-line processing



We'll introduce each of these stages with some examples.

12.1.2 Defining the options

Option processing starts with a definition of each option for the application. This implies that you're required to know in advance all the options possible for an application. This makes inherent sense, because the application itself needs to be aware of the environment and the factors that affect it. You can define options as part of the main application code or as a helper class that creates these options for you. By choosing the latter approach, you can separate application logic from option-creation tasks and let the main application code query the helper classes for the presence or absence of options.

Listing 12.1 shows a very simple application that uses a helper class to define and parse an option. The application code requires that the user be welcomed with a personalized greeting. The application uses a helper class not only to define the user's name as an option, but also to validate and return it.

Listing 12.1 Option definition using a helper class

```
package com.manning.common.chapter12;

public class MainApplicationV1 {
    public static void main(String args[]) {
        System.err.println("Hello " + HelperV1.processArgs(args));
    }
}

class HelperV1 {
    static String processArgs(String args[]) {
        if (args.length != 2 || !args[0].equals("-n")) { ← Define options
            usage();
            System.exit(-1);
        }

        return args[1];
    }

    static void usage() {
        System.err.println("java MainApplicationV1 -n [Your Name]");
    }
}
```

As you may notice, the helper class defines options at the same time it's validating them. For simple applications like this, option definition is almost always done with option validation. Here, the option definition declares that two arguments are required in order to successfully run the `MainApplicationV1` application and that one of them must be `-n`.

As you'll see later, CLI makes the process of option definition more formal by using the `Option` and `Options` classes. Next, we'll look at how to parse options and make them available for use.

12.1.2 Parsing options

Option parsing is the step that accepts, separates, and prepares the options on a command line as defined in the previous stage, for processing by the application code. In listing 12.1, options are parsed in the

Module 13 sample

Understanding and using Chain

13.1 Introducing the Chain component	1
13.2 Using the Chain component.....	2
13.3 Struts and the Chain component	6
13.4 Summary	11
Index	12

Chain of Responsibility is a pattern that allocates the task of handling a request to multiple classes, one after another. Each class attempts to handle the request as best as it can, until a class can handle it in a complete and precise manner. Classes in this chain have no knowledge of the other classes and act in isolation, with only the request parameters available for information and possible modification.

The Chain component from the Commons stable provides an API that you can use to model this Chain of Responsibility pattern in your applications. In this section, we'll take a brief look at this API.

13.1 Introducing the Chain component

Three important concepts in the Chain component define its API: command, chain, and context. We'll discuss these concepts before we introduce some examples.

13.1.1 A unit of work is a command

A *command* is an individual unit of work. Think of a command as a link in a chain of work tasks. However, unlike a link, the command doesn't have any knowledge of the previous and the next commands; nor does it know its position in the chain. The purpose of a command is to do some work and give the result in boolean terms to the calling function.

In Chain, a command is represented by the `Command` interface in the `org.apache.commons.chain` package. This interface defines a single method called `execute` that takes a `context` as a parameter and returns a boolean `true` or `false` to indicate success or failure.

13.1.2 All the commands make a chain

A *chain*, not surprisingly, is a collection of commands in order. Upon receiving a request, the chain starts executing the commands registered with it, in the order they were registered. Each command is executed until a command informs the chain it that it has successfully completed the task, at which point processing stops and no further commands are executed. If no command is successful in completing the task, the chain returns with the response of the last command in the chain or an exception.

The Chain interface represents a chain in this API. There is a concrete implementation class called `ChainBase` in the `org.apache.commons.chain.impl` package. Chains are also executed using the `execute` command, which begins calling the `execute` method of the chain's individual commands until one of them returns `true` or no more commands are left to execute (or an exception is thrown).



13.1.3 Session state is in context

A chain runs in a particular *context*. A context, like a session context in web applications, represents state information. Each command in the chain gets a context, which represents the current state of the chain's variables. Not surprisingly, the context is implemented as a map, thus allowing name-value pairing.

A context is defined by the Context interface. An implementation is provided by the ContextBase class in the `org.apache.commons.chain.impl` package.

Other Chain concepts

In addition to these three concepts, Chain also introduces the idea of a *catalog*, which is defined by the Catalog interface, and which allows Chain and command instances to be identified together using a common key. Also, a *filter* is defined as a special type of command that can perform a post-processing function. The idea is that if a filter special command is part of a chain, and if it's called to perform work (by calling its `execute` method), the chain will also call a `postprocess` method to do some post-processing work after the `execute` method is called—*regardless* of the result of the `execute` method.

Let's see how the Chain component can be used.

13.2 Using the Chain component

Although the Chain component is designed to operate in environments that require complex workflow processing, it isn't tied to a particular implementation. It has been designed to keep its API separate from implementations; it supplies targeted implementations for the Web, Struts, portlets, and JavaServer Faces, but the core of the concept remains in the five interface definitions we talked about earlier. Therefore, in this section, we'll use the main three interfaces to develop a Chain of Responsibility application to illustrate how easy it is to work with this API. Make sure that you download `commons-chain.jar` and have it installed in your CLASSPATH before trying these examples.

The application we'll develop is very simple. The idea is to simulate a workflow situation where a `String` value needs to be pre- and post-processed. This processing alters the `String` to add extra characters. For example, if a `String` has the value, "Chain is great", the value is changed to " {Chain is great" by preprocessing and to " {Chain is great} " by post-processing. The process isn't complete without both operations. Listing 13.1 shows a common base class for this operation.

Listing 13.1 Superclass for processing a String

```
package com.manning.commons.chapter13;

import org.apache.commons.chain.Command;
import org.apache.commons.chain.Context;

public abstract class StringProcessorCommand implements Command {
    public abstract boolean execute(Context context);
    public String getMainStringKey() {
        return this.mainStringKey;
    }
    public void setMainStringKey(String mainStringKey) {
```

Implement
Command
interface

execute
command is
abstract

Module 14 sample

Working with the Logging and Discovery components

14.1 Logging makes me happy	1
14.2 Discovery channel.....	6
14.3 Summary	11
Index	12

This module discusses two Commons components that didn't seem to fit anywhere else, for one reason or another. The Logging and Discovery components have been around for quite a while, and they're included together in this last module so we can introduce them with examples; if your interest is piqued, read on for further information. Let's start with the Logging package.

14.1 Logging makes me happy

Logging is one of the silent components of the Commons library. It's almost ubiquitous, makes minimal fuss, and does a lot of good by providing a consistent and flexible logging apparatus.

The Logging component is essentially a wrapper library that works with existing logging implementations. It provides a transparent working environment for writing log statements from within applications. It doesn't provide the logging logic or the log implementation itself: All it does is discover the underlying logging library using a standard mechanism and supply it to your application.

Let's examine the process that the Logging API adopts to look for the underlying libraries.

14.1.1 Searching for the libraries

The Logging component uses a well-defined process to look for the underlying libraries. But what libraries is the Logging component looking for? Most of the popular ones, as listed here:

- Log4J (<http://jakarta.apache.org/log4j>)
- Avalon logkit (<http://avalon.apache.org/logkit/>)
- JDK 1.4 logging
- Its own SimpleLog and NoOpLog

It isn't difficult to add another logging library to this list, which underlines the usefulness of the Logging component. Moving between the underlying library implementations is easy, if Commons Logging is being used as an abstraction; so, it makes sense to use it rather than an implementation.



The task of looking for these libraries falls on the `LogFactoryImpl` class. As the name suggests, this class is the implementation class for the `LogFactory` interface. Just as this factory is responsible for looking for and selecting the right logging implementation, the `Log` interface is responsible for directing logging calls to the underlying libraries. It has several implementations, one for each of the possible logging libraries.

So what is that well-defined process that looks for the underlying libraries? The steps are listed here:

1. The Logging library looks for the value of a configuration attribute named `org.apache.commons.logging.Log` (or `org.apache.commons.logging.log`). Actually, the `LogFactoryImpl` class looks for this attribute, so you can set this attribute in code by calling `LogFactoryImpl.setAttribute(String className)`.
2. If that attribute isn't found, the `LogFactoryImpl` looks for a system property by the same name.
3. If there is no system property by that name, the factory class tries to load Log4J's `Logger` class and its own `Log` implementation for it (`Log4JLogger`).
4. If Log4J's classes can't be loaded, the factory tries to load the JDK 1.4's logging class (`java.util.logging.Logger`) and its own implementation for it (`Jdk14Logger`).
5. If step 4 fails, the factory looks for JDK 1.3's logging class (`java.util.logging.Logger`) and its own implementation (`Jdk13LumberjackLogger`).
6. Finally, if nothing else is available, the factory defaults to using the `SimpleLog` class.

The value of the `org.apache.commons.logging.Log` attribute or the system property from steps 1 and 2 must be equal to the fully qualified name of the corresponding `Log` implementation that is to be used. As of now, it can be any one of the following values:

```
org.apache.commons.logging.impl.AvalonLogger
org.apache.commons.logging.impl.Jdk13LumberjackLogger
org.apache.commons.logging.impl.Jdk14Logger
org.apache.commons.logging.impl.Log4JLogger
org.apache.commons.logging.impl.LogKitLogger
org.apache.commons.logging.impl.NoOpLog
org.apache.commons.logging.impl.SimpleLog
```

The discovery of a library to choose doesn't happen until a call is made to get a `Log` instance to write to. In the next section, you'll see how to make this happen in code.

14.1.2 Starting the process

One of the easiest ways to start using the Logging component is to put `commons-logging.jar` in your application's `CLASSPATH` and let it try to configure itself. As you've seen in the previous section, if all else fails, Logging resorts to using the `SimpleLog` implementation that writes all messages to `System.err`, the standard error output stream (which is equal to the Standard output stream, unless you change it).

But how does the logging begin? When does the factory start looking for the underlying libraries? The first time, a call is made to the factory for a log to write to anywhere in the code. It's as simple as shown in listing 14.1.

