Chapter 14

Introduction to Collections

So far you've been introduced to only one way of storing a collection of objects — with Java arrays, which are good for storage, but fall short when you need to dynamically add or remove data, or to sort or traverse your collection. There are a number of classes and interfaces in the package `java.util` that are quite handy when multiple instances of some objects have to be co-located in memory. This lesson will introduce you to several of them.

You can find more collections in the `java.util.concurrent` package, but those will be reviewed in Lesson 21 after you become familiar with the concept of multi-threading. Together, the collection classes and interfaces located in `java.util` and `java.util.concurrent` are often called the Java Collection Framework.

Arrays Revisited

Java collection classes enable the storing of handles of related data in one place. Here a handle is a reference to the location of an object in memory. You were introduced to arrays in Lesson 5: Arrays let you store and access a group of variables of the same type. Let's go over the steps you follow to declare and populate an array.

First, declare a variable of the type that matches the types of the objects that will be stored in the array, and reserve enough memory to accommodate all objects. For example, to reserve memory enough for

storing 10 instances of class `Customer` you can declare a variable `cust`, as follows:

```
Customer cust[] = new Customers[10];
```

At this point you've allocated enough space for the storage of 10 handles, not for the actual objects.

Next, create instances of the objects and store their handles in the array. Only the memory addresses of these instances will be stored in the array `cust`:

```
Customer[] cust = new Customer[10];

cust[0] = new Customer("David","Lee");
cust[1] = new Customer("Ringo","Starr");
     ...
cust[9] = new Customer("Lucy","Mann");
```

Now let's give a 15 percent discount to all customers who spent more than $500 in our online store:

```
for (Customer c: cust){
  if (c.getTotalCharges() > 500){
      c.setDiscount(15);
  }
}
```

Note the use of the `for-each` loop here. It safely iterates through this array without trying to access elements beyond its boundaries. You could've used another syntax of the `for` loop to access array elements by index. If a programmer forgot to check the size of the array and tried to access, say, eleven's element like `cust[10].setDiscount(15)`, Java would throw a run-time `ArrayIndexOutOfBoundsException`. Remember, if an array has `n` elements the last element's index is `n -1`.

Classes ArrayList and Vector

Arrays offer the fastest access to the collection of data, but you have to know in advance the number of elements to be stored there. Luckily Java has classes that don't have this restriction, and you can add more elements to a collection during the run time if needed. Classes `Vector` and `ArrayList` belong to this group.

Internally both collections use the array for storage, but when you keep adding elements to these collections they internally increase the size of the array (`ArrayList` increases the size by smaller increments), and as elements are deleted these collections shrink.

In both `Vector` and `ArrayList` you can store only objects — only primitives are allowed. Having said this, keep in mind that Java supports autoboxing (see Lesson 3), and if you'll try to add a primitive to a collection, it'll be automatically converted into he corresponding wrapper object.

You have to pay a price for this convenience: `ArrayList` is a little slower than the array as it needs to do internal array copying to change the collection's size, and `Vector` is even slower because it supports thread synchronization (which will be explained in Lesson 21). But after the introduction of more efficient concurrent collections, `Vector` became less popular and you can achieve its synchronized functionality by using the method `Collections.synchronizedList()` on an `ArrayList` object. All further code samples in this section use only `ArrayList`.

To create and populate an `ArrayList` object you should first instantiate it, and then add instances of other objects to the `ArrayList` by calling the method `add()`:, as in **Listing 14-1**.



Listing 14-1: Populating ArrayList

```
ArrayList customers = new ArrayList();
Customer cust1 = new Customer("David","Lee");
customers.add(cust1);
Customer cust2 = new Customer("Ringo","Starr");
customers.add(cust2);
```

The method `add()` doesn't copy the instance of the object into the `customers` collection, it just stores the memory address of the object being added. The element numbering in `ArrayList` starts with `0`. If you know that an `ArrayList` will store, say, 10 objects, instantiate it with the constructor that will allocate the right amount of memory on creation:

```
ArrayList customers = new ArrayList(10);
```

You can still add more than 10 elements, but JVM will allocate additional memory as needed.

The method `get()` is used to extract a particular element from `ArrayList`. Because `ArrayList` is generic storage for any type of object, the method `get()` returns elements as `Object` data types. It's the responsibility of the programmer to provide proper casting, such as the following:

```
Customer theBestCustomer=(Customer) customers.get(1);
```

To illustrate a possible run-time error that will occur if the casting was not properly done, let's add an object of another type to our `customers` collection from **Listing 14-1**:

```
Order ord = new Order(123, 500, "IBM");
customers.add(ord);
```

The Java compiler will not complain because `ArrayList` can store any objects. At this point we have the elements in `customers` — two customers and one order. As a matter of fact, in Lesson 15 you'll see another way of working with a collection capable of storing objects of unspecified type aka **generics**. I may be jumping ahead here, but the proper (generic) way of declaring collection for storing customers is shown below:

```
ArrayList<Customer> customers = new ArrayList(10);
```

The following code will throw the `IllegalCastException` on the third iteration of the loop:

```
int totalElem = customers.size(); // number of elements
for (int i=0; i< totalElem;i++){
  Customer currentCust = (Customer) customers.get(i);
```

```
    currentCust.doSomething();
  }
```

Listing 14-2 shows how the operator `instanceof` helps you avoid this exception. But before using `instanceof`, see if you can come up with a more elegant solution, as you learned to do in the section "Polymorphism" in Lesson 7. You'll find more samples of working with vectors in Lesson 21.



Listing 14-2: ArrayList and instanceof

```
ArrayList customers = new ArrayList(3);

// The code to populate customers with instances of Customer and Order
// objects is omited for brevity

int totalElem = customers.size();

// Iterate through the list customers and do something with each
//  element of this collection

for (int i=0; i<totalElem;i++){
  Object currElement = customers.get(i);
  if (currElement instanceof Customer){
    Customer currentCust= (Customer)customers.get(i);
    currentCust.doSomething();
  }
  else if (currElement instanceof Order){
    Order currentOrder = (Order) customers.get(i);
    currentOrder.doSomething();
  }
}
```

Collection Interfaces from java.util

A typical collection class implements several interfaces, which represent a well-designed hierarchy. For example, `ArrayList` implements the `List` interface, which extends `Collection`. Just reading the comments in the source code for `java.util` habitats can give you a good idea of what they are for.

`Collection` extends `Iterable`; there are no classes that directly implement this interface. Only its descendent interfaces are implemented. You can use a `for-each` loop (see the "Arrays Revisited" section earlier in this lesson) with classes that implement `Iterable`.

The `List` interface is defined for the ordered collections: `ArrayList`, `Vector`, and `LinkedList`. It allows duplicate elements. For example, the following code snippet will create two elements in `ArrayList`:

```
myArrayList.add("Mary");
myArrayList.add("Mary");
```

The `Set` interface is implemented by collections that don't allow duplicate elements, e.g., `HashSet` and `SortedSet`. For example, the following code snippet will create one element in `HashSet`. The second line will find out that Mary already exists, and won't change it and will return `false`:

```
myHashSet.add("Mary");
myHashSet.add("Mary");
```

The `Map` interface is for storing key/value pairs. A map can't contain duplicate keys and each key can be mapped to only one value (object). You'll see some relevant code examples in the next section.

The `Queue` interface is mainly for collections that require first-in-first-out (FIFO) operation (so-called **priority queues** are the exception). Every new element is added to the tail of the queue and the elements are retrieved from the head of the queue. You can restrict the size of the queue if need be. `LinkedList` is one of the classes that implement the `Queue` interface.

Now let's look at some of the classes that implement these interfaces.

Classes Hashtable and HashMap

The classes `Hashtable` and `HashMap` implement the `Map` interface storing key/value pairs. These classes offer a convenient way of storing and accessing the elements of a collection: by key. You can assign a key to an instance of some Java object and use it as a reference. The collections store objects as key/value pairs. Let's say we need to store instances of the classes `Customer`, `Order`, and `Portfolio` in the same collection. The code snippet from **Listing 14-3** creates these instances first, and then puts them in the collection under some identifiers (keys).



Listing 14-3: Hashtable for key/value pairs

```
Customer cust = new Customer("David", "Lee");
Order ord = new Order(123, 500, "IBM");
Portfolio port = new Portfolio(123);

Hashtable data = New Hashtable();
data.put("Customer", cust);
data.put("Order",ord);
data.put("Portfolio", port);
```

The values in double quotes represent keys by which the objects could be retrieved. In this example the keys are represented by the Java class `String`, but you can use any objects as keys. The keys are selected based on the application needs, e.g. the code in **Listing 14-3** could be written in some order management application.

If you have an idea of how many elements you are planning to store in a `Hashtable`, use another constructor:

```
Hashtable data = new Hashtable(10); // 10-element capacity
```

The method `get()` provides access to these objects via the key. You need to either perform the proper casting as shown below or use **generics** (explained in Lesson 15):

```
Order myOrder = (Order) data.get("Order");
```

The method `size()` returns the number of elements in the `Hashtable`:

```
int totalElem = data.size();
```

Methods `containsKey()` and `containsValue()` help you to find out if the collection contains a specific key or value. You can find an example of `Hashtable` usage in Lesson 31.

The class `HashMap` is similar to `Hashtable`, but it allows `null` as a key or value and is not synchronized. If you are writing code that attempts to ac-

cess the same element concurrently without using multi-threading, use `HashMap`, as it performs faster than `Hashtable`.

To speed up the table lookup, both `HashMap` and `Hashtable` index the data by applying a so-called **hash function** that (based on the contents of the object) generates a **hash code**, one number that represents a large object. There's a slight chance that two different objects will generate the same hash code, but the same object will always produce the same hash code. You can read more about hash functions in Wikipedia at http://en.wikipedia.org/wiki/Hash_function.

Class Properties

Pretty often a desktop application offers you a way to specify and store user preferences such as fonts and colors. This is a use case in which storage of key/value pairs is exactly what's needed. You can store such preferences locally or on remote servers. In Lesson 16 you'll learn how to work with files and other data streams, but from a data structure perspective you'll be dealing with a collection of key/value pairs, such as color=red, font=verdana.

Windows-based applications often store some configurable parameters in the `.ini` files. In general, Java applications store their properties in plain text files, XML files, database tables, and others.

In this section you'll see some code fragments illustrating how the Java class `Properties`, which extends `Hashtable`, can be used to manipulate with properties using key/value pairs. The class `Properties` has one restriction that `Hashtable` does not: Both the key and the value have to be of type `String`. In Lesson 19 you'll see a complete application called MailMan that reads the data needed for sending emails from the `mailman.properties` file, which has the following contents:

```
SmtpServer=mail.xyz.com
to=abc@xyz.com
cc=mary@xyz.com
from=yakov @xyz.com
```

To load this file into the `Properties` object, just define an input stream on this file (see Lesson 16) and call the method `load()`, as shown in **Listing 14-4**. After the file has been loaded into the `Properties` object, each individual property can be obtained with the method `getProperty()`.



Listing 14-4: Reading file mailman.properties into the Properties object

```
Properties prop=new Properties();
FileInputStream in =null;
try{
  new FileInputStream ("mailman.properties");
  prop.load(in);
}catch(Exception e){...}
finally{... in.close();...}

String from = prop.getPropery("from")
String mailServer=prop.getProperty("SmtpServer");
...
```

Java does not have global variables, but as a workaround you can make these properties available to any object in your application by turning them into system properties available from any class in your application:

```
System.setProperties(prop);
```

But keep in mind that the preceding line will also replace the existing system properties, which you may or may not want to do. Now you can get these values from any other class in your application, for example:

```
String mailServer = System.getProperty("SmtpServer");
```

If you decide to store properties in XML files, the class `Properties` offers you the method `loadFromXML()` to read properties and the method `storeToXML()` to store them in a simple XML format.

Enumeration and Iterator

In general, enumerations are sets of items that are related to each other. For example, shipment options or ice cream flavors — such enumerations are supported by the Java keyword `enum` (see Lesson 20). But because we are talking about collections, the meaning of the term **enumeration** is somewhat different. If a collection object implements the interface `Enumeration`, you can traverse its elements sequentially, without even knowing the total number. You just need to obtain the enumeration of all elements and use the methods `hasMoreElements()` and `nextElement()`. For example, to process all elements of `Vector customers` do the following:

```
Vector customers = new Vector();
...
```

```
Enumeration enum=customers.elements();//returns Enumeration

while(enum.hasMoreElements()){
   Customer currentCust = (Customer)enum.nextElement());
   currentCust.doSomething();
}
```

If you're using `ArrayList customers`, the utility class `Collections` comes in handy, as it has a method that returns enumeration:

```
Enumeration enum = Collections.enumeration(customers);
```

You can also obtain the enumeration of a `Hashtable`'s keys or elements. For example:

```
Hashtable data = new Hashtable();
...
// Get the keys
Enumeration enumKeys = data.keys();
while(enum.hasMoreElements()){
...
}
// Get the elements
Enumeration enumElements = data.elements();
...
```

In a way the `Enumeration` interface is similar to `Iterator`, which was introduced as a part of the Java Collections Framework years after. It also offers a standard way to process elements of a collection sequentially. The main difference between the two is that `Enumeration` is a read-only means of traversing a collection, while `Iterator` has a method called `remove()` that enables you to delete unwanted elements of the collection.

`Enumeration` is considered a legacy interface and you should use `Iterator`. For example, you can iterate through the `ArrayList customers` as follows:

```
Iterator iCust = customers.iterator();
while (iCust.hasNext()){
   System.out.println(iCust.next())
}
```

Class LinkedList

Java collection classes differ in how you can retrieve and insert objects. If you need to work with a sequential list of objects and often insert the object into the list, the data structure called **linked list** can fit the bill. Java has a class called `LinkedList` that stores elements so that each contains a reference to the next (aka node). Each element in a **doubly linked list** also contains a reference to the previous element. These features of the doubly-linked class `LinkedList` allow you to create queues (FIFO), and stacks (last-in-first-out or LIFO).

Insertion of a new object inside the list comes down to a simple update of two references: The previous element of the list has to be pointed to the newly inserted object, which will have to include a reference to the next element, if any. Compare this to the complexity of lots of memory allocations and object moving in memory to increase the size of an `ArrayList` or `Vector`, and you'll appreciate the value that linked lists bring to the table. On the other hand, collections that use arrays for the underlying data storage offer random access to the data elements, while linked lists can be processed only sequentially.

You can navigate through the list using the class `ListIterator`, which supports going through the list in both directions via its methods `next()` and `previous()`. **Listing 14-5** shows you an example, in which a standby passenger list is created at the boarding gate of some airline company.



Listing 14-5: LinkedList example

```
import java.util.LinkedList;
import java.util.ListIterator;

public class TestLinkedList {

 public static void main(String[] args) {

  LinkedList passengerList = new LinkedList();

  passengerList.add("Alex Smith");
  passengerList.add("Mary Lou");
  passengerList.add("Sim Monk");

  // Get the list iterator and print every element of the list
  ListIterator iterator = passengerList.listIterator();

  System.out.println(iterator.next());
  System.out.println(iterator.next());
  System.out.println(iterator.next());
 }

}
```

The code in **Listing 14-5** will iterate and print all the objects from the list using `ListIterator`. You might be wondering how the `println()` method knows how to print an object returned by the iterator. It tries to find the method `toString()` defined on the object and call it. In our example the object is a string itself, but in a real-world situation you might need to print objects, and defining the `toString()` method is the right way to do so.

If you use `add()` or `remove()` while iterating through the list, the new element will be either inserted or removed at the iterator's current position.

Class BitSet

The class `BitSet` stores a sequence of bits. It's a pretty efficient class when you need to pass to a program a number of flags that indicate certain conditions. Think of a financial trading application that must be extremely fast. One way to improve the performance is to represent the maximum amount of information in a minimal number of bytes.

Another use case for `BitSet` are programs that send signals with information about the state of a certain device. For example, some vending machines have smart chips that can automatically dial their owner's phone number and send a signal containing status information. Sending a set of flags (bits that are set to `1` or `0`) instead of text or numbers is the most economical way to do this.

The `BitSet` class does not have a size limit and can grow as needed. Depending on which bit is set (e.g., has the value of `1`) the class could indicate the following:

- **Bit 0**: The coin box is empty.
- **Bit 1**: The coin box is half full.
- **Bit 2**: The coin box is full.
- **Bit 3**: The coin box has been removed.
- **Bit 4**: The Coca-Cola row is empty.

One instance of a `BitSet` object carries multiple parameters describing its status. The program that receives this signal could print a nice report, and the owner of this remote machine could decide if he or she needs to send a technician to look at the machine.

The Java class `BitSet` is nothing more than a vector of bits. The following code prepares a signal indicating that the coin box is full and there are no Coca-Cola bottles left.

```
import java.util.BitSet;
class VendingMachineSender {
   public static void main(String args[]){
       BitSet report = new BitSet();
       report.set(2);   // box is full
       report.set(4);   // no Coca Cola
   }
}
```

When the phone call comes in, the callback method `phoneRings()` is invoked and the signal can be decoded like this:

```
import java.util.BitSet;
class VendingMachineListener {
   public void phoneRings(BitSet signal)
      int size = signal.size();

      for (int i=0;i<size;i++){
          if (signal.get(i)){
            switch (i){
               case 0:
                 System.out.println("Box is empty");
                 break;
               case 1:
                 System.out.println("Box is half full");
                 break;
               case 2:
                 System.out.println("Box is full");
                 break;
               // more cases come here
            }
          }
      }
   }
}
```

Try It

Modify the `LinkedList` example from **Listing 14-5** to add an arbitrary object, say the VIP customer after the very first element of the list. You must do this while iterating through the list. When the program is ready it should print the following:

```
Alex Smith
VIP Customer
Mary Lou
Sim Monk
```

Lesson Requirements

You should have Java installed.



You can download the code and resources for this Try It from the book's web page at **www.wrox.com**. You can find them in the Lesson14 folder in the download.

Step-by-Step

1. Create a new Eclipse project called Lesson14.

2. After the first call to `iterator.next()` add the following line:

```
iterator.add("VIP Customer");
```

3. Run the program and observe that it doesn't print "VIP Customer." This happens because the iterator is already positioned after the newly inserted object.

4. Add the line `iterator.previous()` right after the "VIP Customer" to move the iterator one element back.

5. Add one more print statement (otherwise the program won't reach Sim Monk). Compile and run the program. It will print all four elements as requested.

6. Now break the code by changing the line that you added in Step 2 to

`passengerList.add("VIP Customer");`

Run the program now. It'll print the first element of the linked list and then produce a run-time exception:

```
Alex Smith
Exception in thread "main" java.util.ConcurrentModificationException
       at java.util.LinkedList$ListItr.checkForComodification(LinkedList.java:761)
       at java.util.LinkedList$ListItr.next(LinkedList.java:696)
       at TestLinkedList.main(TestLinkedList.java:20)
```

The reason for this concurrent modification exception is that one **thread of execution** was iterating through a collection, and at the same time another thread was trying to modify the underlying collection in an unsafe way. The concept of threads will be introduced in Lesson 20.



Please select Lesson 14 on the DVD with the print book, or watch online at www.wrox.com/go/fainjava to view the video that accompanies this lesson.

O'REILLY

Chapter 15

Introduction to Generics

In the previous lesson you saw an example of a collection that stores objects of different types (see Listing 14-2). During the run time, that program would test the actual type of each object and cast it to an appropriate type — `Customer` or `Order`. If some code adds an element of another (unexpected) data type, this will result in a casting error, `IllegalCastException`.

Starting from Java 5 you can use **generics**, which enables you to use parameterized data types — you can declare an object, collection, or method without specifying a concrete data type, shifting the definition of concrete types to the code that will be using these objects, collections, or methods. And the good part is that by using such generic notation you'll get help from Java compiler, which will not allow you to use objects of the "wrong" types that don't match the declaration. In other words, you can catch improper data types earlier, during the compilation phase.

Generics with Classes

Not only Java methods can have parameters, but classes can have them as well. Consider the `ArrayList` from Listing 14-2, which is a kitchen sink–like storage that can hold pretty much any object. But if you add the parameterized type `Customer` in angle brackets to the declaration of the `customers` collection (see **Listing 15-1**), any attempt to place an `Order` object there will generate the following compiler error:

```
The method add(Customer) in the type ArrayList<Customer> is not applicable
for the arguments (Order).
```

Think of it this way: `ArrayList` can be used to store any objects, and using generics allows you to put a constraint on the types of objects allowed in a specific instance of `ArrayList`. This is an example of a parameterized object, which is just one use for generics.



Listing 15-1: Using generics in the collection

```
import java.util.ArrayList;
import java.util.ArrayList;

public class TestGenericCollection {

       public static void main(String[] args) {

              ArrayList<Customer> customers = new ArrayList<Customer>();
              Customer cust1 = new Customer("David","Lee");
              customers.add(cust1);
              Customer cust2 = new Customer("Ringo","Starr");
              customers.add(cust2);
              Order ord1= new Order();

              customers.add(ord1); // Compiler error

        }
}
```

Getting an error during compilation is better than getting run-time cast exceptions. What makes the `ArrayList` class capable of rejecting the unwanted data types? Open the source code of the `ArrayList` itself (press-

ing F3 in Eclipse shows the source code of any class or interface, if available). It starts as follows:

```
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable
```

This magic `<E>` after the class name tells the Java compiler that some types of elements will be stored in this class, but which ones will remain unknown until a concrete instance of `ArrayList` is created. In **Listing 15-1** the **type parameter** `<Customer>` replaces `<E>`, and from now on the collection `customers` will accept only instances of `Customer` objects.

I'd like to stress that this `<E>` notation is used only during the declaration of the type. The code in **Listing 15-1** does not include `<E>`. The compiler replaces `<E>` with `Customer` and erases the parameterized data type. This process is known as **type erasure** — it's done for compatibility with code written in older versions of Java that didn't have generics.

Now you can simplify the code from Listing 14-2 by removing casting (see **Listing 15-2**). Why? Because with generics, when the compiler sees a specific type, it automatically generates the bytecode, which performs casting internally! That's why you don't even need to cast the data returned by the method `get(i)` from `Object` to `Customer` any longer. Besides, you're guaranteed that the collection `customers` will have only `Customer` instances.



Listing 15-2: Iterating through customers without casting

```
ArrayList<Customer> customers = new ArrayList<Customer>();
```

```
// The code to populate customers is omited for brevity

// Iterate through the list customers and do something with each
//  element of this collection. No casting required.

for (Customer c: customers){
       c.doSomething();
}
```

Defining Generics

If you'll be creating your own class for storing other objects, you can use any letter(s) in angle brackets to declare that your class will use parameterized types. Some developers use `<E>` for element; some prefer `<T>` for type. Such a letter will be replaced by a concrete type when used with generics' notation during concrete variable declaration. Open the source code of the Java class `Hashtable` and you'll see `<K,V>`, which stands for key and value:

```
public class Hashtable<K,V> extends Dictionary<K,V>
    implements Map<K,V>, Cloneable, java.io.Serializable {...}
```

Again, what types will be used for storing keys and values is decided when the `Hashtable` is used. You can use a parameterized type for declaring variables where you'd use regular data types. **Listing 15-3** shows a fragment from the source code of the interface `java.util.List`. The interface declaration uses `<E>` as a data type.



Listing 15-3: Fragment from java.util.List interface

```
package java.util;

public interface List<E> extends Collection<E> {

    Iterator<E> iterator();

    <T> T[] toArray(T[] a);

    boolean add(E e);

    boolean containsAll(Collection<?> c);

    boolean addAll(Collection<? extends E> c);

    boolean addAll(int index, Collection<? extends E> c);

    boolean removeAll(Collection<?> c);

    E set(int index, E element);

    void add(int index, E element);

    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);

    List<E> subList(int fromIndex, int toIndex);
}
```

Wild Cards

Listing 15-3 contains question marks that represent unknown types. It'll be easier to explain them with an example. Let's turn the `for` loop from **Listing 15-2** into a function. In Eclipse, highlight the code of the `for` loop, click the right mouse button, and select Refactor ⇒ Extract Method. In the pop-up window enter the method name `processCustomers` and click OK.



Listing 15-4: Refactored class TestGenericCollection

```
import java.util.ArrayList;
import java.util.Hashtable;
import java.util.List;

public class TestGenericCollection {

	public static void main(String[] args) {

		ArrayList<Customer> customers = new ArrayList<Customer>();
		Customer cust1 = new Customer("David","Lee");
		customers.add(cust1);
		Customer cust2 = new Customer("Ringo","Starr");
		customers.add(cust2);
		Order ord1= new Order();
		//customers.add(ord1); // Compiler error

		// Iterate through the list customers and do something with each
		//  element of this collection. No casting required.

		processData(customers);

	}

	private static void processData(ArrayList<Customer> customers) {
		for (Customer c: customers){
			c.doSomething();
		}
	}

}
```

What if you want to make the method `processData()` more generic and useful not only for a collection of `Customer` objects but for others too? In the pre-generics era you'd be using `instanceof` and writing something similar to **Listing 15-5**.



Listing 15-5: Back to casting

```
private static void processData(ArrayList data) {

               for (Object o: data){
                    ((Customer) o).doSomething();
               }
     }
```

But now, armed with the new knowledge, you can change the signature of the method process `Data` to

```
private static void processData(ArrayList<Object> data){...}
```

Unfortunately, this solution won't work, because there is no such thing as inheritance of parameterized types. In other words, even though the class `Customer` is a subclass of `Object`, this relationship does not apply to `<Customer>` and `<Object>`. This is when the question mark that represents an unknown type becomes handy. The next step in making `processData()` more generic is to change the method signature to the following:

```
private static void processData(ArrayList<?> data){...}
```

Using such a method signature is different from simply declaring the method argument data of type `ArrayList`, which would require casting, as in **Listing 15-5**. With the wild-card notation you state, "At this point the type of data is not known, but whenever some code uses the method `processData()` it'll be known and properly compiled so casting won't be needed."

The next challenge you face is to compile the code calling the method `doSomething()` on the objects of unknown types.

Bounded Wild Cards

Bounded wild cards enable you to specify generic types that are not completely unknown, but rather "somewhat unknown." You can say, for example, that the method `processData()` from the previous section can work only with the objects that extend a specific class. Consider an example in **Listing 15-6**, which declares two classes — `Customer` and `Order` — that extend `Data`.



Listing 15-6: Customer and Order extending Data

```
public abstract class Data {
       abstract void doSomething();
}

public class Customer extends Data{
       private String fName;
      private String lName;

    public Customer (String fName, String lName){
       this.fName=fName;
       this.lName=lName;
    }

       void doSomething(){
            System.out.println("Customer " + fName + " " +lName +
". In doSomething()");
       }
}

public class Order extends Data{
      void doSomething(){
             // Do something specific to Order processing
```

```
        }
}
```

You can rewrite the method `processData()` using the bounded wild card `<? extends Data>`, as in the following example:

```
private static void processData(ArrayList<? extends Data> data) {
       for (Data d: data){
          d.doSomething();
       }
}
```

If I didn't show you the code in **Listing 15-6** and asked, "What does the bounded type `<? extends Data >` mean?", the right answer would have been, "It represents any subclass of the class `Data` or any class that implements the interface `Data`." The word `extends` may sound a little misleading in the latter case, but the designers of Java generics decided to keep this keyword for both classes and interfaces.

If you want to use multiple bounds, they should be separated by ampersands:

```
<? extends Data & Payable >
```

You can achieve yet another interesting effect by using the keyword `super` in the generics. For example, the next declaration means that you are allowed to use any superclass of `Data`:

```
<? super Data & Payable >
```

The `super` keyword is quite handy when you need to ensure that arguments of a method belong to the same inheritance hierarchy. For example, the method `doSomething()` can be used for processing such lists as `Person` and `RetiredEmployee` (assuming that `RetiredEmployee` is a subclass of `Employee`).

```
<T>  void doSomething(ArrayList<? super T> all, ArrayList<? extends T> best){
   //do something
}
```

Generic Methods

While declaring a method you can either predefine data types for its arguments and return values or use generics. For example, the method `toArray()` from **Listing 15-3** starts with a declaration of a new parameterized type (`<T>` in that case), which has to be placed in angle brackets right before the return type in the method signature. The very fact that a method declares a new type makes it generic. the following declaration of the `toArray()`method takes an array of objects of type `T` and returns an array of `T` objects:

```
<T> T[] toArray(T[] a);
```

If you have an `ArrayList` of integers, you can declare and convert it to an array as follows:

```
ArrayList<Integer> myNumericList = new ArrayList<Integer>();
...
Integer myNumericArray = new Integer[myNumericList.size()];
myNumericArray = myNumericList.toArray();
```

If you need to use the same method `toArray()` with a list of customers, the data type `<T>` magically transforms (by compiler) into the `Customer` type:

```
ArrayList<Customer> myCustomerList = new ArrayList< Customer >();
...
Customer myCustomerArray = new Customer [myCustomerList.size()];
myCustomerArray = myCustomerList.toArray();
```

As in examples from the “Bounded Wild Cards” section, you are allowed to put constraints on the type. For example, you can restrict the `toArray()` method to work only with types that implement the `Comparable` interface:

```
<T extends Comparable> T[] toArray(T[] a);
```

What to Read Next

This book is a concise tutorial that doesn’t have room to cover Java generics in greater detail. If you’d like to get a deeper understanding of this topic, refer to the book Java Generics and Collections, 2006, written by Maurice Naftalin and Philip Wadler and published by O’Reilly, ISBN 0-596-52775-6.

Bruce Eckel devoted almost a hundred pages to generics in the fourth edition of his book Thinking in Java, 2006, published by Prentice Hall, ISBN 0-131-87248-6.

Try It

Create a simple program that uses generics with the class `RetiredEmployee` (which extends the class `Employee`) from Listing 7-2. Write a generic method that will accept a collection of `RetiredEmployee` objects and copy it into a collection of `Employee` objects.

Lesson Requirements

You should have Java installed.



You can download the code and resources for this Try It from the book's web page at **www.wrox.com**. You can find them in the Lesson15 folder in the download.

Step-by-Step

1. Create a new Eclipse project called Lesson 15.

2. Create a class called `RetiredEmployee` that extends `Employee`.

3. Create an executable Java class, `TestGenericMethod`, that will accept a List of `RetiredEmployee` objects and copy it into a `List` of `Employee` objects. This method should print on the system console the name of each `Employee` from the resulting collection.

4. Run the `TestGenericMethod` program and observe the printed names.



Please select Lesson 15 on the DVD with the print book, or watch online at **www.wrox.com/go/fainjava** to view the video that accompanies this lesson.

Chapter 16

Working with Streams

Most programs work with some kind of data, which could be stored in a local database, on a remote computer, or in a file located on your disk. Java has a concept of working with **streams of data**. You can say that a Java program reads sequences of bytes from an input stream (or writes into an output stream): byte after byte, character after character, primitive after primitive. Accordingly, Java defines various types of classes supporting streams, for example `InputStream` or `OutputStream`. There are classes specifically meant for reading character streams such as `Reader` and `Writer`. `DataInputStream` and `DataOutputStream` can read and write Java primitives, and to work with files you may consider such classes as `FileInputStream` and `FileReader`.

Classes that work with streams are located in two packages: `java.io` and `java.nio`. Classes from the former implement blocking of input/output (I/O): When bytes are being read/written by a process, they become unavailable for other threads of execution. The latter package offers non-blocking I/O with improved performance. In this lesson you'll be dealing with streams defined in `java.io`.

Before deciding which Java class to use for I/O in each particular case, you need to understand what kind of data is coming from the stream in question. But no matter what you select, your code will need to perform three operations:

1. Open a stream that points at a specific data source: a file, a socket, a URL, and so on.

2. Read or write data from/to this stream.

3. Close the stream.

If a Java program uses a third-party program, such as database management systems (DBMS), you won't need to program streams directly — the database drivers or object-relational mapping framework is all you need. In this lesson you'll see examples of performing I/O operations with different streams.

Byte Streams

A program can read or write any file one byte at a time with the help of one of the subclasses of `InputStream` or `OutputStream` respectively. The example in **Listing 16-1** shows how to use the class `FileInputStream` to read a file named `abc.dat`. This code snippet reads and prints each byte's value:



Listing 16-1: Using FileInputStream

```
FileInputStream myFile = null;
  try {
            myFile = new  FileInputStream("abc.dat");
            boolean eof = false;

            while (!eof) {
                int byteValue = myFile.read();
                System.out.print(byteValue + " ");
                if (byteValue  == -1){
                    eof = true;
                }
            }
          // myFile.close(); // do not do it here!!!
```

```
    } catch (IOException e) {
            System.out.println("Could not read file: " + e.toString());
    } finally{
       if (myFile !=null){
          try{
              myFile.close();
          } catch (Exception e1){
               e1.printStackTrace();
          }
       }
    }
 }
```

Because the code in **Listing 16-1** doesn't specify the directory where `abc.dat` is located, the program will try to find this file in the current directory, which is the root directory of the Eclipse project. At any given time you can easily find out the current directory programmatically by calling the method `System.getProperty("user.dir")`.

The output of this program will be a sequence of numbers, which represents the codes of the characters located in the file. For example, if `abc.dat` contains the text "This is a test file," the output on the system console will look like this:

```
84 104 105 115 32 105 115 32 97 32 116 101 115 116 32 102 105 108 101 -1
```

When you are reading with `FileInputStream`, the end of the file is represented by a negative one, and this is how you know when to stop. The code in **Listing 16-1** checks for `-1` and sets the `boolean` variable `eof` to `false` to finish the loop. Note that the stream is closed in the clause `finally`. Do not call the method `close()` inside the `try/catch` block right after the file reading is done. In case of an exception during the file read, the program will jump over the `close()` statement into the `catch` section, and the stream will never be closed!

The code fragment in **Listing 16-2** writes into a file called `xyz.dat` using the class `FileOutputStream`.



Listing 16-2: Using FileOutputStream

```
// byte values are represented by integers from 0 to 255
  int somedata[]= {56,230,123,43,11,37};
  FileOutputStream myFile = null;
  try {
     myFile = new  FileOutputStream("xyz.dat");
     for (int i = 0; i <somedata.length; i++){
        file.write(data[i]);
     }
 } catch (IOException e) {
       System.out.println("Could not write to a  file: " + e.toString());
  } finally{
      if (myFile !=null){
        try{
          myFile.close();
        } catch (Exception e1){
          e1.printStackTrace();
        }
      }
  }
}
```

Buffered Streams

The code in the previous section was reading and writing one byte at a time. In general, disk access is much slower than the processing performed in memory; that's why it's not a good idea to access disk a thousand times to read a file of 1000 bytes. To minimize the number of times the disk is accessed, Java provides so-called **buffers**, which serve as reservoirs of data.

The class `BufferedInputStream` works as a middleman between `FileInputStream` and the file itself. It reads a big chunk of bytes from a file into memory in one shot, and the `FileInputStream` object then reads single bytes from there, which is memory-to-memory operations. `BufferedOutputStream` works similarly with the class `FileOutputStream`. The main idea here is to minimize disk access.

Buffered streams are not changing the type of reading — they just make reading more efficient. Think of it this way: A program performs stream chaining (or stream piping) to connect streams, just as pipes are connected in plumbing. **Listing 16-3** shows an example in which a file is read so the data from `FileInputStream` will fill `BufferedInputStream` before processing.



Listing 16-3: Chaining FileInputStream with BufferedInputStream

```
FileInputStream myFile = null;
BufferedInputStream buff =null;
  try {
     myFile = new  FileInputStream("abc.dat");
     buff = new BufferedInputStream(myFile);
            boolean eof = false;
            while (!eof) {
                int byteValue = buff.read();
                System.out.print(byteValue + " ");
                if (byteValue  == -1)
                    eof = true;
           }
} catch (IOException e) { ...  }
 finally{
if (myFile !=null){
        try{
          buff.close();
          myFile.close();
        } catch (Exception e1){
          e1.printStackTrace();
        }
```

```
        }
    }
```

While reading a stream with the help of `BufferedInputStream` you watch for the end-of-file character. But when you write a stream via `BufferedOutputStream` with method `write()`, you need to do one extra step: Call the method `flush()` before closing the buffered stream. This ensures that none of the buffered bytes "get stuck" and that all are written out to the underlying output stream.

You might be wondering, how large is the buffer? While the default buffer size varies depending on the OS, you can control it using a two-argument constructor. For example, to set the buffer size to 5000 bytes instantiate the buffered stream as follows:

```
BufferedInputStream buff = new BufferedInputStream(myFile, 5000);
```

Character Streams

The text in Java is represented as a set of `char` values (two-byte characters), which are based on the Unicode Standard. The Java classes `FileReader` and `FileWriter` were specifically created to work with text files, but they work only with default character encoding and don't handle localization properly.

The recommended way is to pipe the class `InputStreamReader` with specified encoding and the `FileInputStream`. The class `InputStreamReader` reads bytes and decodes them into characters using a specified `CharSet`. Each JVM has a default charset, which can be specified during the JVM start-up and depends on the locale. Some of the standard charsets are US-ASCII, UTF-8, and UTF-16.

Listing 16-4 reads bytes from a text file and converts them from UTF-8 encoding into Unicode to return results as a `String`. For efficiency, the reading is piped with the `BufferReader`, which reads text from the stream buffering characters. Note that this code uses `StringBuffer` that usually works faster than `String` when it comes to performing text manipulations.



Listing 16-4: Reading text files

```
StringBuffer buffer = new StringBuffer();
    try {
       FileInputStream myFile = new FileInputStream("abc.txt");
       InputStreamReader inputStreamReader =
                  new InputStreamReader(myFile, "UTF8");
      Reader reader = new BufferedReader(inputStreamReader);

      int ch;  // the code of one character

      while ((ch = reader.read()) > -1) {
             buffer.append((char)ch);
      }

      buffer.toString();

    } catch (IOException e) {
        ...
    }
```

For writing characters to a file, pipe `FileOutputStream` and `OutputStreamWriter`. For efficiency, use `BufferedWriter`, for example

```
try{
  String myAddress = "123 Broadway, New York, NY 10011";
```

```
    FileOutputStream myFile = new FileOutputStream("abc.txt");

    Writer out
       = new BufferedWriter(new OutputStreamWriter(myFile, "UTF8"));
    out.write(myAddress);

  } catch(IOException e){
        ...
  }
```

Listing 16-5 shows yet another version of the tax calculation program. This time I've added a text file, `states.txt`, that includes states that will be used to populate a drop-down box, `chStates`. My file is located in the root directory of the Eclipse project Lesson16, and it looks like this:

```
New York
New Jersey
Florida
California
```

The program in **Listing 16-5** requires a class, `Tax`, that you can borrow from Lesson 4. Make sure that it has the method `calcTax()`.



Listing 16-5: Bringing together Swing and streams

```
import java.awt.event.*;
import javax.swing.*;
import java.awt.GridLayout;
import java.io.FileInputStream;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
```

```
import java.util.Vector;

public class TaxGuiFile extends JFrame implements ActionListener {
    JLabel lblGrIncome;
    JTextField txtGrossIncome = new JTextField(15);
    JLabel lblDependents=new JLabel("Number of Dependents:");
    JTextField txtDependents = new JTextField(2);
    JLabel lblState = new JLabel("State: ");

    //Define a data model for the ComboBox chState
    Vector states = new Vector(50);

    //Create a combobox to get data from the model
    JComboBox chState = new JComboBox(states);

    JLabel lblTax = new JLabel("State Tax: ");
    JTextField txtStateTax = new JTextField(10);
    JButton bGo = new JButton("Go");
    JButton bReset = new JButton("Reset");

    TaxGuiFile() {
        lblGrIncome = new JLabel("Gross Income: ");
        GridLayout gr = new GridLayout(5,2,1,1);
        setLayout(gr);

        add(lblGrIncome);
        add(txtGrossIncome);
        add(lblDependents);
        add(txtDependents);
        add(lblState);
        add(chState);
        add(lblTax);
        add(txtStateTax);
        add(bGo);
        add(bReset);

        // Populate states from a file
        populateStates();

        chState.setSelectedIndex(0);

        txtStateTax.setEditable(false);

        bGo.addActionListener(this);
        bReset.addActionListener(this);

        // Define, instantiate and register a WindowAdapter
        // to process windowClosing Event of this frame
```

```
        this.addWindowListener(new WindowAdapter() {
           public void windowClosing(WindowEvent e) {
                 System.out.println("Good bye!");
                  System.exit(0);
           }});
     }

     public void actionPerformed(ActionEvent evt) {
        Object source = evt.getSource();
         if (source == bGo ){
           // The Button Go processing
             try{
               int grossInc =Integer.parseInt(txtGrossIncome.getText());
               int dependents = Integer.parseInt(txtDependents.getText());
               String state = (String)chState.getSelectedItem();

               Tax tax=new Tax(dependents,state,grossInc);
               String sTax =Double.toString(tax.calcTax());
               txtStateTax.setText(sTax);
             }catch(NumberFormatException e){
                 txtStateTax.setText("Non-Numeric Data");
             }catch (Exception e){
                txtStateTax.setText(e.getMessage());
             }
          }
          else if (source == bReset ){
             // The Button Reset processing
              txtGrossIncome.setText("");
              txtDependents.setText("");
         chState.setSelectedIndex(0);
              txtStateTax.setText("");
              }
     }
    // The code below will read the file states.txt and
    // populate the dropdown chStates
     private void populateStates(){

         states.add("Select State");

         FileInputStream myFile=null;
         InputStreamReader inputStreamReader;
         BufferedReader reader = null;

         try {
          myFile = new FileInputStream("states.txt");
          inputStreamReader =
                       new InputStreamReader(myFile, "UTF8");
           reader = new BufferedReader(inputStreamReader);
```

```
            String nextLine;
            boolean eof = false;
            while (!eof) {
                 nextLine = reader.readLine();
                  if (nextLine == null){
                    eof = true;
                  } else {
                    // Populate the model
                    states.add(nextLine);
                  }
            }

          }catch (IOException e){
              txtStateTax.setText("Can't read states.txt");
          }
          finally{
            // Closing the streams
            try{
                if (myFile !=null){
                 reader.close();
                 myFile.close();
                 }
            }catch(IOException e){
               e.printStackTrace();
            }
          }
      }

      public static void main(String args[]){
         TaxGuiFile taxFrame = new TaxGuiFile();
         taxFrame.setSize(400,150);
         taxFrame.setVisible(true);
      }
  }
```

The code in **Listing 16-5** reads the content of the file `states.txt` and populates a collection — a `Vector` with states. This collection plays the role of a data model for the combo box states. I used a constructor of `JComboBox` that takes a data model as an argument. This Swing component knows how to display the content of its data model.

This is an example of the implementation of the so-called **MVC (model-view-controller)** design pattern, which promotes the separa-

tion of data and UI. `JComboBox` plays the role of a view, the vector is a model, and the user works as controller when she selects a particular state and the view has to be updated. The output of the program from **Listing 16-5** is shown in **Figure 16-1**.

Figure 16-1



Data Streams

If you are expecting to work with a stream of a known data structure (e.g., two integers, three floats, and a double) use either `DataInputStream` for reading or `DataOutputStream` for writing. A method, `readInt()`, of `DataInputStream` will read the whole integer number (four bytes) at once, and `readLong()` will get you a long number (eight bytes).

The class `DataInputStream` is yet another filter that can be connected to another stream. **Listing 16-6** has an example of how you can "build a pipe" from the following pieces:

`FileInputStream` ⇒ `BufferedInputStream` ⇒ `DataInputStream`



Listing 16-6: Using DataInputStream

```
FileInputStream myFile = new FileInputStream("myData.dat");
BufferedInputStream buff = new BufferedInputStream(myFile);
```

```
DataInputStream data = new  DataInputStream(buff);

try {
   int num1 = data.readInt();
   int num2 = data.readInt();
   float num2 = data.readFloat();
   float num3 = data.readFloat();
   float num4 = data.readFloat();
   double num5 = data.readDouble();
} catch (EOFException eof) {...}
```

In this example `FileInputStream` opens the file `myData.dat` for reading, `BufferedInputStream` makes the read more efficient, and `DataInputStream` extracts from the buffer two integers, three floats, and a double. The assumption here is that the file `myData.dat` contains exactly these data types and in the specified order. Such a file could have been created with the help of `DataOutputStream`, which allows you to write primitive Java data types to a stream in a portable way. It has a variety of methods to choose from: `writeInt()`, `writeByte()`, `writeFloat()`, and so on.

The Class File

The class `File` allows you to rename or delete a file, perform an existence check, create a directory, and more. If you need this functionality, start by creating an instance of this class:

```
File myFile = new File("abc.txt");
```

This line does not create a file; it just creates in memory an instance of the class `File` that's ready to perform its action on the file `abc.txt`. If you want to create a physical file, use the method `createNewFile()` defined in the class `File`. Here's the list of some methods of the class `File`:

- `createNewFile()`: Creates a new, empty file named according to the file name used during the file instantiation. Creates a new file only if a file with this name does not exist.
- `delete()`: Deletes file or directory.
- `renameTo()`: Renames a file.
- `length()`: Returns the length of the file in bytes.
- `exists()`: Tests whether the file with the specified name exists.
- `list()`: Returns an array of strings containing file and directory.
- `lastModified()`: Returns the time that the file was last modified.
- `mkDir()`: Creates a directory.

The next code fragment checks for the existence of the file `customers.txt.bak`, deletes it if it is found, and then renames the file `customers.txt` to `customers.txt.bak`:

```
File file = new File("customers.txt");
File backup = new File("customers.txt.bak");
if (backup.exists()){
       backup.delete();
}
file.renameTo(backup);
```

Try It

Write a program that will read a `.zip` archive file and print on the system console the list of files included in the zip archive. Do a little research about the class `java.util.zip.ZipInputStream` and use it together with `FileInputStream`. Read about the class `ZipEntry` too.

Lesson Requirements

You should have Java installed.



You can download the code and resources for this Try It from the book's web page at **www.wrox.com**. You can find them in the Lesson16 folder in the download.

Step-by-Step

1. Create a new Eclipse project called Lesson16.

2. Copy any `.zip` file into its root directory.

3. Open `FileInputStream` and connect it with `ZipInputStream`.

4. Write a loop that will use the method `getNextEntry()` from `ZipInputStream`. This method reads the `ZipEntry`, if any, and positions the stream at the beginning of the entry data.

5. Call the function `getName()` on each `ZipEntry` instance found.

6. Print the entry name on the system console.

7. Close the entry inside the loop.

8. Run the program and observe that it properly prints the file names from the selected `.zip` file.

9. If you want to learn how to create `.zip` files from Java, read about `ZipOutputStream`.



Please select Lesson 16 on the DVD with the print book, or watch online at **www.wrox.com/go/fainjava** to view the video that accompanies this lesson.

Chapter 17

Java Serialization

Imagine a building that, with a push of a button, can be turned into a pile of construction materials and the possessions of its residents. Push another button and the building is re-created in its original form in a different location. This is what Java serialization is about. By "pushing the serialize button" you turn an instance of an object into a pile of bytes, and "pushing the deserialize button" magically re-creates the object.

Wikipedia defines the term serialization as

the process of converting a data structure or object into a sequence of bits so that it can be stored in a file or memory buffer, or transmitted across a network connection link to be "resurrected" later in the same or another computer environment. When the resulting series of bits is reread according to the serialization format, it can be used to create a semantically identical clone of the original object… This process of serializing an object is also called deflating or marshalling an object. The opposite operation, extracting a data structure from a series of bytes, is deserialization (which is also called inflating or unmarshalling).

In Lesson 16 you became familiar with streams that deal with single bytes, characters, Java primitives, and text. Now you'll see why and how to write objects into streams, or how to serialize Java objects.

Consider the following scenario: `ClassA` creates an instance of the object `Employee`, which has the fields `firstName`, `lastName`, `address`, `hireDate`, `salary`, etc. The values of these fields (that is, the object's state) have to be saved in a stream. Later on `ClassB`, which needs these data, somehow has to re-create the object `Employee` in memory. The instances of `ClassA` and `ClassB` may live in two different JVMs running on different computers.

Sure enough, you can remember the order and the data types of `firstName`, `lastName`, `address`, `hireDate`, and `salary`, and loop through the fields performing the output with `DataOutputStream`.

Here's another use case: Your application has a Preferences menu where the user can select fonts, colors, and UI controls that should be displayed on the opening view. To support such functionality, the Preferences panel creates an instance of a class, `UserPreferences`, with 50 configurable properties (some of which are of custom data types) that have to be saved in a local file. On the next start of the application the previously saved data should be read from this file with the re-creation of the `UserPreferences` object.

Writing a manual procedure that uses, say, `DataOutputStream` on each primitive property and then recursively performs the same operation for each non-primitive type is tedious. Besides, this code would need to be changed each time the properties of class `UserPreferences` change.

You can achieve the same result in a more elegant way, not one property at a time, but one object at a time, with the help of such streams as `ObjectOutputStream` and `ObjectInputStream`. This process is known as **Java serialization**, which enables you to save objects in a stream in one shot.

`ClassA` will serialize `Employee`, and `ClassB` will **deserialize** (reconstruct) this object. To support this mode the class `Employee` has to implement the `Serializable` interface, as shown in **Listing 17-1**.



Listing 17-1: Serializable class Employee

```
class Employee implements java.io.Serializable{
     String lName;
     String  fName;
     double salary;

}
```

The interface `Serializable` does not define any methods, so the class `Employee` has nothing to implement. If an interface doesn't declare any methods it's called a **marker interface**.

Two use cases described in this section represent the main uses for Java serialization — distribution of the Java objects between different JVMs and object persistence for future reuse. Often serialization is implemented internally by various Java frameworks to pass objects between tiers of a distributed application that runs on multiple servers and client computers.

Some Java frameworks use serialization to load or unload objects to free the JVM's memory (these are known as **passivation** and **activation** of objects, respectively). Later in the book you'll learn about RMI and EJB, which use serialization a lot. But even in regular business application development you may need to serialize and deserialize application-specific objects. This process comes down to reading from and writing into streams, namely `java.io.ObjectInputStream` and `java.io.ObjectOutputStream`, respectively.

The Class ObjectOutputStream

To serialize an object perform the following actions:

1. Open an output stream.

2. Chain it with `ObjectOutputStream`.

3. Call the method `writeObject()`, providing the instance of a `Serializable` object as an argument.

4. Close the stream.

Listing 17-2 shows a code sample that serializes an instance of the `Employee` object into the file `c:\practicalJava\BestEmployee.ser`. This code first opens `FileOutputStream` and then chains it with `ObjectOutputStream`. The method `writeObject()` performs the serialization of the `Employee` instance in one shot.



Listing 17-2: Serializing an Employee object into a file

```
import java.io.*;

class ClassA {

  public static void main(String args[]){
       Employee emp = new Employee();
       emp.lName = "John";
       emp.fName = "Smith";
       emp.salary = 50000;

       FileOutputStream fOut=null;
       ObjectOutputStream oOut=null;

   try{
     fOut = new  FileOutputStream("c:\\practicalJava\\BestEmployee.ser");

     oOut = new ObjectOutputStream(fOut);

     oOut.writeObject(emp);  //serializing employee
```

```
    } catch (IOException e){

        // close the streams
        try{
           oOut.flush();
           oOut.close();
           fOut.close();
        }catch (IOException ioe){
                       ioe.printStackTrace();
        }
    }

    System.out.println("The Employee object has been serialized into " +
                    "c:\\practicalJava\\BestEmployee.ser");
  }
}
```

All Java primitive data types are serializable. All member variables of the serializable class must be either Java primitives or reference variables pointing to objects that are also serializable. The class `Employee` from **Listing 17-1** contains only serializable data types. But if a class has properties of custom data types, each of them has to be serializable. The `Employee` class from **Listing 17-3** can be serialized only if the class `PromotionHistory` (and all data types it uses) was declared with `implements Serializable`.



Listing 17-3: Class Employee with custom data types

```
class Employee implements java.io.Serializable{
     String lName;
     String  fName;
     double salary;

     PromotionHistory promos; // A custom data type
```

```
}
```

If you do not want to serialize some sensitive information, such as salary, declare this variable using the keyword `transient`. If you declare the property salary of the class `Employee` with the `transient` qualifier, its value won't be serialized.

```
transient double salary;
```

Typically, you declare a variable as `transient` either if it contains some sensitive information or if its value is useless in the destination stream. Suppose a Java class that runs on the server has database connectivity variables. If such a class gets serialized to the client's machine, sending database credentials may be not only useless, but unsafe. So such variables have to be declared as `transient`. One more reason to use the `transient` keyword is to serialize an object that includes properties of non-serializable types.

The Class ObjectInputStream

To deserialize an object perform the following steps:

1. Open an input stream.

2. Chain it with the `ObjectInputStream`.

3. Call the method `readObject()` and cast the returned object to the class that is being deserialized.

4. Close the stream.

Listing 17-4 shows a code sample that reads the previously serialized file `BestEmployee.ser`. This code opens a `FileInputStream` that points at `BestEmployee.ser` and then chains the `FileInputStream` with `ObjectInputStream`. The method `readObject()` resurrects the instance of the `Employee` object in memory.



Listing 17-4: Deserializing Employee from file

```
import java.io.*;
class ClassB {

  public static void main(String args[]){

    FileInputStream fIn=null;
    ObjectInputStream oIn=null;

    try{
     fIn = new  FileInputStream("c:\\practicalJava\\BestEmployee.ser");

     oIn = new ObjectInputStream(fIn);

     Employee bestEemp=(Employee)oIn.readObject();

   } catch (ClassNotFoundException cnf){
             cnf.printStackTrace();
   } catch (IOException e){

       try{
         oIn.close();
         fIn.close();
       }catch (IOException ioe){
         ioe.printStackTrace();
       }
   }

   System.out.println("The Employee  object has been deserialized from " +
                              " c:\\practicalJava\\BestEmployee.ser");
```

```
    }
  }
```

Note that the class that deserializes an object has to have access to its declaration or the `ClassNotFoundException` will be thrown. During the process of deserialization all `transient` variables will be initialized with the default values for their type. For example, `int` variables will have a value of `0`.

Keep in mind that class variables with longer names produce larger footprints when the object is serialized. In time-critical applications this may be important. Imagine a Wall Street trading system in which each order is presented as an object, `TradeOrder`, with 50 properties, and you need to send a thousand of these objects over the network using serialization. Simply shortening each variable name by one character can reduce the network traffic by almost 50 KB! You'll learn how to open the network streams in Lesson 18.

The Interface Externalizable

Continuing with the example of the trade order, the knowledge of the values of all 50 fields from the class `TradeOrder` may be required to support certain functionality of the application, but when the program has to send a buy or sell request to the stock exchange, only ten properties may be needed. This raises a question — can the process of serialization be customized so only some of the object properties will be serialized?

The method `writeObject()` of `ObjectOutputStream` sends all the object's properties into a stream. But if you want to have more control over what is being serialized and decrease the footprint of your object, implement the `Externalizable` interface, which is a subclass of `Serializable`.

This interface defines two methods: `readExternal()` and `writeExternal()`. These methods have to be written by you to implement serialization of only

the selected properties. **Listing 17-5** shows a fragment of a class, `Employee2`, that implements `Externalizable`. It has several properties, but in a certain scenario only `id` and `salary` have to be serialized.



Listing 17-5: Externalizable version of Employee

```
class Employee2 implements java.io.Externalizable {
       String lName;
       String fName;
       String address;
       Date hireDate;
       int id;
       double salary;

   public void writeExternal(ObjectOutputStream stream)
                             throws java.io.IOException {
    // Serializing only the salary and id
    stream.writeDouble(salary);
    stream.writeInt(id);
   }

   public void readExternal(ObjectInputStream stream)
                             throws java.io.IOException {
     salary = readDouble();  // Order or reads must be the
                             // same as the order of writes
     id  = readInt();
   }

}
```

Note that each property of the class is individually serialized and the methods `writeExternal()` and `readExternal()` must write and read properties, respectively, in the same order. The class `EmpProcessor` from **Listing 17-6** shows how to **externalize** the object `Employee2`.



Listing 17-6: Turning an object into an array of bytes

```
import java.io.*;
import java.util.Date;

public class EmpProcessor {

      public static void main(String[] args) {

           Employee2 emp = new Employee2();
           emp.fName = "John";
           emp.lName = "Smith";
           emp.salary = 50000;
           emp.address = "12 main street";
           emp.hireDate = new Date();
           emp.id=123;

           FileOutputStream fOut=null;
           ObjectOutputStream oOut=null;

             try{
               fOut= new FileOutputStream(
                  "c:\\practicalJava\\NewEmployee2.ser");

               oOut = new ObjectOutputStream(fOut);
               emp.writeExternal(oOut);  //serializing employee

            System.out.println("An employee is serialized into " +
                        " c:\\practicalJava\\NewEmployee2.ser");

             }catch(IOException e){
                        e.printStackTrace();
             }finally{
                   try {
                             oOut.flush();
                             oOut.close();
                             fOut.close();
                   } catch (IOException e1) {
                             e1.printStackTrace();
                   }
             }
```

```
        }
    }
```

You had to write a little more code to implement the `Externalizable` interface than to implement the `Serializable` interface, but the size of the file `NewEmployee2.ser` will be substantially smaller. First of all, you serialized the values of only two properties, and files created with the `Externalizable` interface contain only data, while files created with `Serializable` also contain class metadata that includes properties' names. The code snippet in **Listing 17-7** shows you how to re-create the externalized object `Employee2`.



Listing 17-7: Re-creating externalized object

```
fIn= new FileInputStream("c:\\practicalJava\\NewEmployee2.ser");
oIn = new ObjectInputStream(fIn);

Employee2 emp = new Employee2();
emp.readExternal(oIn);
```

Class Versioning

Imagine that a program, ClassA, serializes the `Employee` object from **Listing 17-1** into a file on Mary's computer. Two days later Mary starts another program, ClassB, that offers a download of its new version with long-awaited features. After upgrade, ClassB starts generating errors, which are caused by the fact that the new upgrade includes a modified version of the class `Employee` that now has one property with a different data type than what exists in memory. The upgrade also includes one new property that wasn't in the previous version of the `Employee` object.

Now the deserialization process will try to ignore the new property but will fail because of the mismatched property types between the serialized and in-memory versions of `Employee`. Serialization may also fail because of a change in the inheritance tree of `Employee`.

During serialization, JVM automatically calculates a special value: the **serial version unique ID**, which is based on the properties of the serializable object, the class name, the implemented interfaces, and the signatures of non-private methods. If you are curious to see how this number looks for your class, run the program serialver (it's located in the `bin` directory of your Java install), providing the name of your class as a command-line argument.

But if your class explicitly defines and initializes a `static final` variable called `serialVersionUID`, Java will use your value instead of trying to generate one. For example,

```
Public static final  serialVersionUID = 123;
```

Now, if you keep the value of this variable in the new version of `Employee` the same as in the old one, you'll have some freedom to add more methods to this class, and JVM will assume that both classes have the same version. If a new version has added a public field, the deserialization process will ignore it. If a new version has removed a field, the deserialized version will still have it initialized with a default value. But if you change the data type for the public field, the deserialization process will fail anyway.

Serializing into Byte Arrays

An interesting effect will be produced if you serialize your objects into an in-memory array of bytes — `byte[]`. This can be the easiest way of

creating a sort of memory blob that can be exchanged among different VMs.

The syntax of such serialization is pretty straightforward. Let's assume that you have a class called `XYZ` that implements `Serializable` and contains all the elements of your report in the proper format. To prepare a byte array from it, write the code in **Listing 17-8**.



Listing 17-8: Turning an object into an array of bytes

```
XYZ myXyz = new XYZ();

// Code to fill myXyz with the content goes here

//...

ByteArrayOutputStream baOut = new ByteArrayOutputStream(5000);
ObjectOutputStream oOut = new ObjectOutputStream(
                                new BufferedOutputStream(baOut));

//Here comes the serialization part

oOut.writeObject(myXyz);
oOut.flush();

// create a byte array from the stream

byte[] xyzAsByteArray = baOut.toByteArray();
oOut.close();
```

Another convenient use for serializing into a byte array is **object cloning**, which is the creation of an exact copy of an object instance. Even though the class `Object`, the root of all classes, includes the method `clone()`, it works only with objects that implement the `Cloneable` marker interface; otherwise the cloning will fail. Things may get more compli-

cated when an object contains instances of other objects: You need to program a **deep copy** of the object.

Serialization into a byte array with immediate deserialization creates a deep copy of the object in no time. After executing the code from **Listing 17-9** you'll get two identical objects in memory. The variable `bestEmployee` points at one of them, and the variable `cloneOfBestEmployee` points at another.



Listing 17-9: Object cloning with serialization

```
Employee bestEmployee = new Employee();

//Serialize into byte array
ByteArrayOutputStream baOut = new ByteArrayOutputStream();
ObjectOutputStream oOut = new ObjectOutputStream(baOut);
oos.writeObject(bestEmployee);

//Deserialize from byte array
ByteArrayInputStream baIn = new ByteArrayInputStream(baOut.toByteArray());
ObjectInputStream oIn = new ObjectInputStream(baIn);
Employee cloneOfBestEmployee = (Employee) oin.readObject();
```

Try It

Create a Java Swing program called `MyCustomizableGUI` that will enable the user to specify her preferences, such as background color, font family, and size. Selected values should be assigned to properties of the serializable class `UserPreferences` and stored in the file `preferences.ser`. Each time `MyCustomizableGUI` is started it should determine if the file `preferences.ser` exists. If the file does exist, `MyCustomizableGUI` should deserialize it and apply previously saved preferences to the GUI.

Lesson Requirements

You should have Java installed.



You can download the code and resources for this Try It from the book's web page at **www.wrox.com**. You can find them in the Lesson17 folder in the download.

Step-by-Step

1. Create a new Eclipse project and name it Lesson17.

2. Create an executable Swing class called `MyCustomizableGUI` with a text field and a User Preferences button.

3. Program the button to open a new window Preferences (based on `JDialog`), that has three drop-downs (aka `JComboBox`), and Save and Cancel buttons. The first drop-down will allow the user to select a color, the second a font, and the third a font size.

4. Create a serializable class, `UserPreferences`, that will remember the user's selections. When the user has made her choices, the Save button has to serialize the instance of `UserPreferences` into a file named `preferences.ser`.

5. Each time `MyCustomizableGUI` starts it has to determine if the file `preferences.ser` exists, deserialize it if so, and apply the appropriate color as the background color of the window `MyCustomizableGUI`. The font preferences should be applied to the text field.

6. Run the program to ensure that you can change and save the preferences and that they are properly applied to the GUI.



Please select Lesson 17 on the DVD with the print book, or watch online at **www.wrox.com/go/fainjava** to view the video that accompanies this lesson.

Chapter 18

Network Programming

Computers connected to a network can communicate with each other only if they agree on the rules of communication, called **protocols,** that define how to request the data, if the data should be sent in pieces, how to acknowledge received data, if the connection between two computers should remain open, and so on. TCP/IP, UDP/IP, FTP, and HTTP are some examples of network protocols.

Local area network (LAN) refers to a computer network connecting devices in a small area — the same office or house, or a rack. Interconnected computers located farther apart or that belong to different companies are part of a **wide area network (WAN)**. The Internet consists of millions of networks and individual devices. When connected networks belong to the same organization it is referred to as an **intranet.** For security reasons intranets are shielded from the rest of the world by special software called **firewalls.**

The World Wide Web (WWW) uses **uniform resource locators (URLs)** to identify online resources. For example, the following URL says that there is (or will be generated on the fly) a file called `training.html` located at the remote host known as mycompany.com, and that the program should use the HTTP protocol to request this file. It also states that this request has to be sent via port 80.

```
www.mycompany.com:80/training.html
```

The hostname must be unique and it is automatically converted to the IP address of the physical server by your **Internet service provider (ISP)**, aka your hosting company. The IP address is a group of four numbers, for example 122.65.98.11. Most of the individuals connected to the Internet are getting so-called **dynamic** (not **permanent**) IP addresses assigned to their home computers, but for a small extra fee you can request a static IP address that can be assigned to any server located in your basement, office, or garage. In the enterprises, network computers usually get static (permanent) IP addresses. For individual use, it's sufficient to have a dynamically assigned IP address as long as your ISP can find your server by resolving a domain name to the current IP address.

Finding a resource online is somewhat similar to finding a person by his or her address. The role of an IP address is similar to the role of a street

number of a building, and a port plays the role of an apartment number in that building. Many people can live in the same building, just as many programs can run on the same server. A port is simply a unique number assigned to a server program running on the machine.

Multiple Java technologies exist for providing data exchange among computers in a network. Java provides classes for network programming in the package `java.net`. This lesson shows you how to read data from the Internet using the class `URL` as well as direct socket-to-socket programming. Lesson 19 is about sending e-mails with JavaMail API. Lessons 26 through 37 introduce you to other technologies that you can use over the network: Java Servlets, RMI, EJB, and JMS.

Reading Data from the Internet

You learned in Lesson 16 that to read local file streams, a program has to specify the file's location, such as `c:\practice\training.html`. The same holds true for reading remote files — the only difference is that you open the stream over the network. Java has a class, `java.net.URL`, that will help you to connect to a remote computer on the Internet and get access to a resource there, provided that it's not protected from the public. First, create an instance of the `URL` of your resource:

```
try{
  URL xyz = new URL("http://www.xyz.com:80/training.html");
    ...
}
catch(MalformedURLException e){
      e.printStackTrace();
}
```

The `MalformedURLException` will be thrown if an invalid URL has been specified, for example if you typed `htp` instead of `http` or included extra spaces. If the `MalformedURLException` is thrown, it does not indicate that the remote machine has problems — just check the spelling of your URL.

Creation of the `URL` object does not establish a connection with the remote machine; you'll still need to open a stream and read it. Perform the following steps to read a file from the Internet:

1. Create an instance of the class `URL`.

2. Create an instance of the `URLConnection` class and open a connection using the `URL` from Step 1.

3. Get a reference to an input stream of this object by calling the method `URLConnection.getInputStream()`.

4. Read the data from the stream. Use a buffered reader to speed up the reading process.

While using streams over the networks you'll have to handle possible I/O exceptions the way you did while reading the local files. The server you are trying to connect to has to be up-and-running and, if you're using HTTP-based protocols, a special software — web server — has to be "listening to" the specified port on the server. By default, web servers are listening to all HTTP requests on port 80 and to secure HTTPS requests directed to port 443.

The program in **Listing 18-1** reads the content of the existing or generated file `index.html` from **google.com** and prints this content on the system console. For you to test this program your computer has to be connected to the Internet. The program may not work properly if your connection goes through a firewall.



Listing 18-1: Reading the content of index.html from google.com

```
import java.net.*;
import java.io.*;

public class WebSiteReader {

  public static void main(String args[]){

       String nextLine;
       URL url = null;
       URLConnection urlConn = null;

       InputStreamReader  inStream = null;
       BufferedReader buff = null;

       try{

         // index.html is a default URL's  file name
         url  = new URL("http://www.google.com" );
         urlConn = url.openConnection();

         inStream = new InputStreamReader(urlConn.getInputStream(), "UTF8");

         buff  = new BufferedReader(inStream);


         // Read and print the lines from index.html
         while (true){

             nextLine =buff.readLine();

             if (nextLine !=null){
                System.out.println(nextLine);
             }else{
               break;
             }
         }
```

```
       } catch(MalformedURLException e){
          System.out.println("Please check the URL:" + e.toString() );
       } catch(IOException  e1){

          System.out.println("Can't read  from the Internet: "+ e1.toString() );
      } finally{
            if (inStream != null){
              try{
                  inStream.close();
                  buff.close();
              } catch(IOException e1){
                  System.out.println("Can't close the streams: "+ e1.getMessage());
              }
            }
      }
  }
}
```

The code in **Listing 18-1** creates an instance of the class `URL`, then gets a reference to an instance of `URLConnection` to open a connection with the stream, and, finally, opens `InputStreamReader`, which is chained with `BufferedReader`. Run this program and you'll see the output shown in **Listing 18-2**.



Listing 18-2: The fragment of console output shown after google.com is read

```
<!doctype html><html><head><meta http-equiv="content-type" content="text/html;
 charset=ISO-8859-1"><title>Google</title><script>
window.google={kEI:"BHVpTNWrGoGclgeNrczDBA",kEXPI:"25976,26060",
kCSI:{e:"25976,26060",ei:"BHVpTNWrGoGclgeNrczDBA",expi:"25976,26060"},
ml:function(){},kHL:"en",time:function(){return(new Date).getTime()},log:function(b,d,c){var a=new Image,e=goog
a.onerror=(a.onload=(a.onabort=function(){delete g[f]}));g[f]=a;c=c||"/gen_204?atyp=i&ct="+b+"&cad="+d+"&zx="+g
a.src=c;e.li=f+1},lc:[],li:0,Toolbelt:{}};

window.google.sn="webhp";window.google.timers={load:{t:{start:(new Date)
.getTime()}}};try{}catch(u){}window.google.jsrt_kill=1;
...
</scri
pt>
```

The class `WebSiteReader` explicitly creates the object `URLConnection`. Strictly speaking, you could achieve the same result by using the class `URL`:

```
URL url = new URL("http://www.google.com");
InputStream in = url.getInputStream();
Buff= new BufferedReader(new InputStreamReader(in));
```

The reason you may consider using the `URLConnection` class is that it could give you some additional control over the I/O process. For example, by

calling its method `setDoOutput(true)` you specify that this Java program is intended to write to the remote URL too. In the case of HTTP connections, this will also implicitly set the type of request to `POST` (see Lesson 27). The method `useCaches()` of `URLConnection` also allows you to specify whether the protocol can use a cached object or should always request a fresh copy. In general, if you are planning to write Java programs that will work using HTTP protocol, use the class `HttpURLConnection`, which supports HTTP-specific features, such as processing header fields, getting HTTP response codes, setting request properties, and so on.

Connecting through HTTP Proxy Servers

For security reasons, most enterprises use firewalls (see http://en.wikipedia.org/wiki/Firewall_%28computing%29) to block unauthorized access to their internal networks. As a result their employees can't directly reach the outside Internet world (or even some internal servers), but go through HTTP proxy servers. Check the settings of your Internet browser to see if you are also sitting behind a firewall, and find out the hostname and port number of the proxy server if you are. Usually, web browsers store proxy parameters under the Advanced tabs of their Settings or Preferences menus.

If your browser has downloaded a page containing a Java applet, the latter knows the parameters of the proxy servers and can make requests to the remote servers through the firewall. But a regular Java application should specify the `http.proxyHost` and `http.proxyPort` parameters to "drill a hole" in the firewall. For example, if the name of your proxy server is proxy.mycompany.com and it runs on port 8080, the following two lines should be added to your Java application that needs to connect to the Internet:

```
System.setProperty("http.proxyHost","http://proxy.mycompany.com");
System.setProperty("http.proxyPort", 8080);
```

If you do not want to hardcode these values, pass them to your program from the command line:

java -Dhttp.proxyHost=http://proxy.mycompany.com

–Dhttp.proxyPort=8080 WebSiteReader

The other option for programmatically specifying proxy parameters is to do it via the class `java.net.Proxy`. The code for the same proxy server parameter would look like this (you can replace the name of the server with an IP address):

```
Proxy myProxy = new Proxy(Proxy.Type.HTTP,
                    new InetSocketAddress ("http://proxy.mycompany.com", 8080);
url  = new URL("http://www.google.com/index.html" );
urlConn = url.openConnection(myProxy);
```

How to Download Files from the Internet

Combine the class URL with the reading files techniques and you should be able to download practically any unprotected file (such as images, music, and binary files) from the Internet. The trick is in opening the file stream properly. **Listing 18-3** shows the code of a Java class, `FileDownload`, which takes the URL and the destination (local) file name as command-line arguments, connects to this resource, and downloads it into a local file.



Listing 18-3: Downloading an arbitrary file from the Internet

```
import java.io.FileOutputStream;
import java.io.InputStream;
import java.net.URL;
import java.net.URLConnection;

class FileDownload{
   public static void main(String args[]){
    if (args.length!=2){
      System.out.println(
        "Proper Usage: java FileDownload URL OutputFileName");
      System.exit(0);
    }

  InputStream in=null;
  FileOutputStream fOut=null;
  try{
    URL remoteFile=new URL(args[0]);
    URLConnection fileStream=remoteFile.openConnection();
  // Open output and input streams
    fOut=new FileOutputStream(args[1]);
    in=fileStream.getInputStream();

 // Save the file
    int data;
    while((data=in.read())!=-1){
         fOut.write(data);
    }
} catch (Exception e){
   e.printStackTrace();
  } finally{
       System.out.println("The file " + args[0] +
           " has been downloaded successfully as " + args[1]);
     try{
       in.close();
       fOut.flush();
       fOut.close();
```

```
      } catch(Exception e){e.printStackTrace();}
    }
  }
}
```

Note how this `FileDownload` program starts by checking the number of provided command parameters: If the number is anything but two, the program prints an error message and quits. Here's an example of how you can run this program to download a W-4 tax form from the Internal Revenue Service website:

java FileDownload **http://www.irs.gov/pub/irs-pdf/fw4.pdf w4form.pdf**

The Stock Quote Program

In this section you'll learn how to write a program that can read stock market price quotes from the Internet. There are many Internet sites providing such quotes; the Internet portal Yahoo! is one of them.

Visit **http://finance.yahoo.com**, enter the symbol of any stock, MOT for example, and press the Get Quotes button. You'll see the pricing information about Motorola, Inc. — a fragment of this web page is depicted on **Figure 18-1**.

Figure 18-1



But what's more interesting for our goal is the URL that is displayed in the Web browser:

```
http://finance.yahoo.com/q?s=MOT
```

Right-click this web page and select View Source from the pop-up menu to see the HTML contents of this page; you'll see lots of HTML tags, and the information about MOT will be buried somewhere deep inside. Modify the line in the class `WebSiteReader` from **Listing 18-1** to have it print the content of this page on the system console:

```
url  = new URL("http://finance.yahoo.com/q?s=MOT");
```

You can also store the whole page in a Java `String` variable instead of printing the lines. Just modify the `while` loop in **Listing 18-1**:

```
String theWholePage;
String txt;
while (txt =buff.readLine() != null ){
     theWholePage=theWholePage + txt;
 }
```

If you add some smart tokenizing (splitting into parts based on the specified tokens as in **Listing 18-4**) of `theWholePage`, to get rid of all HTML tags and everything but Last Trade info, you can create your own little Stock Quote program. While this approach is useful to sharpen your tokenizing skills, it may not be the best solution, especially if Yahoo! changes the wording it uses on this page. That's why we'll be using another URL that provides stock quotes in a cleaner **comma-separated values (CSV)** format. Here's the URL that should be used for the symbol MOT:

```
http://quote.yahoo.com/d/quotes.csv?s=MOT&f=sl1d1t1c1ohgv&e=.csv
```

This URL produces a string that includes the stock symbol, last trade, date and time of the price quote, earning per share (EPS), opening price, day's range, and volume. Compare the CSB line below with the data shown in **Figure 18-1** — they are the same (the date is not shown on **Figure 18-1**).

```
"MOT",7.81,"8/16/2010","4:00PM",0.17,7.63,7.86,7.58,25934265
```

Now the task of tokenizing the entire web page comes down to parsing a short CSV line. The `StockQuote` class from **Listing 18-4** does exactly this: It accepts the stock symbol from the command line, gets the data from Yahoo!, tokenizes the received CSV line, and prints the price quote on the console.



Listing 18-4: Printing stock quotes

```
import java.net.*;
import java.io.*;
import java.util.StringTokenizer;

public class StockQuote {

   static void printStockQuote(String symbol){

      String csvString;
       URL url = null;
       URLConnection urlConn = null;
       InputStreamReader  inStream = null;
       BufferedReader buff = null;

       try{
           url  = new URL("http://quote.yahoo.com/d/quotes.csv?s="
                   + symbol + "&f=sl1d1t1c1ohgv&e=.csv" );

           urlConn = url.openConnection();
           inStream = new InputStreamReader(urlConn.getInputStream());

           buff  = new BufferedReader(inStream);

          // get the quote as a csv string
           csvString =buff.readLine();

          // parse the csv string
              StringTokenizer tokenizer = new StringTokenizer(csvString, ",");

              String ticker = tokenizer.nextToken();
              String price  = tokenizer.nextToken();
              String tradeDate = tokenizer.nextToken();
              String tradeTime = tokenizer.nextToken();

              System.out.println("Symbol: " + ticker +
                " Price: " + price + " Date: "  + tradeDate
                + " Time: " + tradeTime);

     } catch(MalformedURLException e){
         System.out.println("Please check the spelling of "
                             + "the URL: " + e.toString() );
     } catch(IOException  e1){
       System.out.println("Can't read from the Internet: " + e1.toString() );
     }finally{
         try{
           inStream.close();
           buff.close();
         }catch(Exception e){
           System.out.println("StockQuote: can't close streams" +
                                                    e.getMessage());
         }
     }
   }

  public static void main(String args[]){

       if (args.length==0){
          System.out.println("Sample Usage: java StockQuote IBM");
          System.exit(0);
       }

       printStockQuote(args[0]);
  }

}
```