1

Key Java Language Features and Libraries

The past two major releases of the JDK have seen some significant changes. JDK 5 introduced new features at the language level, something that has not happened since Java was first released. Some of the most significant features added to the language are generics (parameterized types), enumerations, and metadata. With JDK 6, one of the biggest changes is the inclusion of a lightweight database known as Derby, which is from the Apache Database project.

The first half of this chapter introduces Derby and reviews the new language features from JDK 5. The second half of this chapter details certain key utility packages in the `java.util` branch of the class library that are useful for professional programming in Java.

Introducing Derby

New to Java 6 is a lightweight database called Derby, a product of the Apache Database project. Derby is a transactional, relational database and provides a small footprint on disk. As of the July 13, 2006 release of the Java 6 JDK, Derby is installed (by default) in `C:\Program Files\Java\jdk1.6.0\db` and includes the core libraries, demonstration programs, and an example database. Derby started its life as CloudScape, a product IBM acquired with Informix. In 2004, IBM decided to open source CloudScape and it became an incubator project under the Apache Software Foundation with the name Derby. The real benefits to using Derby are that it has minimal administration needs and a small footprint. The databases are small on disk, roughly 2MB for a basic database. The fact that administration is minimal allows you, as a developer, to easily create and use databases in code. This speeds up development. Deployment is made that much easier because Derby supports storage of a database archived in a JAR file, allowing you to simply distribute the JAR file.

Because Derby is an involved topic, this section serves only to introduce Derby and its features, using the included command-line tool and a brief exploration of using the JDBC driver. Derby is revisited later in this book.

Using Derby

As mentioned, Derby is automatically installed as part of the JDK. Derby provides a command-line tool called `ij`, which is an abbreviation for interactive JDBC scripting tool. This tool provides a way to connect to and manipulate Derby databases. You must have the following JAR files in your classpath before using this tool. The `derby.jar` file contains the JDBC drivers, and `derbytools.jar` contains the `ij` tool itself:

```
c:\Program Files\Java\jdk1.6.0\db\lib\derby.jar
c:\Program Files\Java\jdk1.6.0\db\lib\derbytools.jar
```

After your classpath is configured, start the tool and connect to the example database (toursdb) included with Derby:

```
c:\>java org.apache.derby.tools.ij
ij version 10.2
ij> connect 'jdbc:derby:c:\Program Files\Java\jdk1.6.0\db\demo\databases\toursdb';
ij>
```

Don't forget the semicolon at the end of a command. If you leave this off, `ij` may seem like it's processing a command, but it isn't. This provides for ease of entering multiline commands such as creating tables or complicated select statements. These semicolons are confined to `ij` and are not passed to the database.

The tool works much as you would expect it to, such as issuing a select statement to retrieve a partial listing of data from the countries table (a table that is part of the example toursdb):

```
ij> select * from countries where country like 'A%';
COUNTRY                   |C&|REGION
--------------------------------------------------------
Afghanistan               |AF|Asia
Albania                   |AL|Europe
Algeria                   |DZ|North Africa
American Samoa            |AS|Pacific Islands
Angola                    |AO|Africa
Argentina                 |AR|South America
Armenia                   |AM|Europe
Australia                 |AU|Australia and New Zealand
Austria                   |AT|Europe
Azerbaijan                |AZ|Central Asia
```

To create a new database from `ij`, include the parameter `create=true` to the connection string. Because you're already connected to the toursdb, first disconnect. The `select` statement proves you're disconnected. Then issue the new `connect` statement:

```
ij> disconnect;
ij> select * from countries;
IJ ERROR: Unable to establish connection
ij> connect 'jdbc:derby:DerbyTestDB;create=true';
ij>
```

The database name (in this case, DerbyTestDB) is created as a subdirectory of the directory where you started the `ij` tool. The database appears on disk in the directory `C:\DerbyTestDB`. Exploring this directory is strictly for curiosity's sake — you should never have to modify any file in this directory, including the `service.properties` file that may seem tempting to play with. The creation of a database also creates a `derby.log` file at the same level as the DerbyTestDB, so the file in this case is `C:\derby.log`. This log file is an error log and is useful to check to get more information about the inevitable problems that arise during real development. If you create multiple databases they will share this log file.

Now that you have a new database, create a table, insert some data, and query it:

```
ij> create table zipcodes(zipcode varchar(5), city varchar(20), state varchar(2));
0 rows inserted/updated/deleted
ij> insert into zipcodes values ('22030', 'Fairfax', 'VA');
1 row inserted/updated/deleted
ij> insert into zipcodes values ('22003', 'Annandale', 'VA');
1 row inserted/updated/deleted
ij> insert into zipcodes values ('90210', 'Beverly Hills', 'CA');
1 row inserted/updated/deleted
ij> select * from zipcodes;
ZIPC&|CITY                |STA&
-------------------------------
22030|Fairfax             |VA
22003|Annandale           |VA
90210|Beverly Hills       |CA

3 rows selected
ij>
```

By default, auto-commit is `on` from the `ij` tool, so you don't have to issue the `commit;` command to save changes to the database. You can control auto-commit by issuing the command "autocommit on;" or "autocommit off;" Type "exit;" to exit from the `ij` tool.

Now that you have seen the basics of using the `ij` tool, look at an example of querying your newly created database from a Java program using the JDBC driver. Because the standard JDBC mechanism is used, there are no surprises with the import statements:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

import java.util.Properties;

public class DerbyTestDBClient {
   public static void main(String[] args)
   {
      DerbyTestDBClient testClient = new DerbyTestDBClient();

      testClient.showZipCodes();
   }
```

The `showZipCodes` method actually opens the connection and performs the query. The driver used is `org.apache.derby.jdbc.EmbeddedDriver`. Derby also includes a `ClientDriver` for connecting to Derby in network mode, where Derby runs a network server providing for a client/server approach to using Derby:

```
      public void showZipCodes()
      {
         try {
            String driver = "org.apache.derby.jdbc.EmbeddedDriver";

            Class.forName(driver).newInstance();
            Connection conn = null;
            conn = DriverManager.getConnection("jdbc:derby:DerbyTestDB");
            Statement s = conn.createStatement();
            ResultSet rs = s.executeQuery("SELECT city, state, zipcode
                                           FROM zipcodes");

            while(rs.next()) {
               System.out.println("City   : " + rs.getString(1));
               System.out.println("State  : " + rs.getString(2));
               System.out.println("Zipcode: " + rs.getString(3));
               System.out.println();
            }

            rs.close();
            s.close();
            conn.close();
         } catch(Exception e) {
            System.out.println("Exception: " + e);
            e.printStackTrace();
         }
      }
   }
```

Here's the output from the previous code:

```
   c:\>java DerbyTestDBClient
   City   : Fairfax
   State  : VA
   Zipcode: 22030

   City   : Annandale
   State  : VA
   Zipcode: 22003

   City   : Beverly Hills
   State  : CA
   Zipcode: 90210
```

Derby is a thriving project and continues to implement more features. You can keep an eye on its development at `http://db.apache.org/derby`.

Language Features Added in Java 5

Several useful syntactic elements were introduced in Java 5. All these features are supported by an updated compiler, and all translate to already defined Java bytecode, meaning that virtual machines can execute these features with no need for an update:

- **Generics:** A way to make classes type-safe that are written to work on any arbitrary object type, such as narrowing an instance of a collection to hold a specific object type and eliminating the need to cast objects when taking an object out of the collection.
- **Enhanced `for` loop:** A cleaner and less error-prone version of the `for` loop for use with iterators.
- **Variable arguments:** Support for passing an arbitrary number of parameters to a method.
- **Boxing/unboxing:** Direct language support for automatic conversion between primitive types and their reference types (such as `int` and `Integer`).
- **Type-safe enumerations:** Clean syntax for defining and using enumerations, supported at the language level.
- **Static import:** Ability to access static members from a class without need to qualify them with a class name.
- **Metadata:** Coupled with new tools developed by third-party companies, saves developers the effort of writing boilerplate code by automatically generating the code.

These features update the Java language to include many constructs developers are used to in other languages. They make writing Java code easier, cleaner, and faster. Even if you choose not to take advantage of these features, familiarity with them is vital to read and maintain code written by other developers.

Generics

Java 5 introduced generics, also known as parameterized types. Generics allow you to write a class that can operate on any type but that specific type is not specified until declaration of an instance of the class. Because this type is not specified as part of the class definition, the class becomes generic, gaining the ability to work on any type specified. The most obvious example, and a great use of generics, is the collection classes. The `ArrayList` class, for example, was written to hold, simply, `Object`. This means objects lose their type when added to the `ArrayList` and a cast is needed when accessing an element of the `ArrayList`. However, code that uses a generic version of the `ArrayList` can say "I want this `ArrayList` to hold only `Strings`." This adds additional type-safety to Java because, if anything other than a `String` is added to the collection, the compiler will catch it. This also means that a cast is no longer needed when accessing elements — the compiler knows it only holds `Strings` and will produce an error if the elements are treated as anything other than a `String`. Specifying `String` as the parameterized type is as easy as placing the type in angle brackets:

```
ArrayList<String> listOfStrings; // <TYPE_NAME> is new to the syntax
String stringObject;

listOfStrings = new ArrayList<String>(); // <TYPE_NAME> is new to the syntax
listOfStrings.add(new String("Test string")); // Can only pass in String objects
stringObject = listOfStrings.get(0); // no cast required
```

Generics are also known as *parameterized types* where a type is the parameter. As can be seen in the previous example, `String` is the *formal* type parameter. This same parameterized type must be used when instantiating the parameterized type.

Because one of the goals of the new language features in Java 5 was to not change the Java instruction set, generics are, basically, syntactic sugar. When accessing elements of the `ArrayList`, the compiler automatically inserts the casts that you now don't have to write. It's also possible to use the primitive data types as a parameterized type, but realize that these incur boxing/unboxing costs because they are implicitly converted to and from `Object`. Nonetheless, there are benefits in increased type-safety and increased program readability.

Type Erasure

A generic type in Java is compiled to a single class file. There aren't separate versions of the generic type for each formal parameterized type. The implementation of generics utilizes type erasure, which means the actual parameterized type is reduced to `Object`. Strangely, the decision to use erasure, although requiring no bytecode changes, hobbles the generics mechanism in its determination to maintain strong typing, as you'll soon see.

By way of example, the following code will not compile:

```
interface Shape {
   void draw();
}

class Square implements Shape {
   public String name;

   public Square()
   {
      name = "Square";
   }

   public void draw()
   {
      System.out.println("Drawing square");
   }
}

public class ErasureExample {
   public static <T> void drawShape(T shape)
   {
      shape.draw();
   }

   public static void main(String args[])
   {
      Square square = new Square();
   }
}
```

The compiler issues the following error:

```
ErasureExample.java:23: cannot find symbol
symbol  : method draw()
location: class java.lang.Object
      shape.draw();
           ^
1 error
```

If you replace the `drawShape` method with the following, the compiler is now happy to compile the program:

```
public static <T> void drawShape(T shape)
{
   System.out.println("Hashcode: " + shape.hashCode());
}
```

Why this discrepancy? It's the result of type erasure. The `hashCode` method belongs to `Object`, however the draw method belongs only to objects of type `Shape`. This little experiment demonstrates that the parameterized type is actually reduced to an `Object`. The next example shows how this relates to use of a generic class with different parameterized types.

Start with a new generic class to hold a data item of arbitrary type:

```
public class CustomHolder<E>
{
   E storedItem;

   public E getItem()
   {
      return(storedItem);
   }

   public void putItem(E item)
   {
      System.out.println("Adding data of type " + item.getClass().getName());
      storedItem = item;
   }
}
```

By convention, single letters are used for formal type parameters, usually E for element and T for type. Add a main method to this class:

```
public static void main(String args[])
{
   CustomHolder<String> stringHolder = new CustomHolder<String>();
   CustomHolder<Object> objectHolder = new CustomHolder<Object>();
   String str = new String("test string");
   String str2;

   stringHolder.putItem(str);
```

```
        objectHolder.putItem(str);

        str2 = stringHolder.getItem();
        //str2 = objectHolder.getItem();
    }
```

Look at the last two lines. Retrieving an element from `stringHolder` and assigning it to a string is fine. However, if you uncomment the second line, which tries to access the same string in the `objectHolder`, you get the following compiler error.

```
c:\>javac CustomHolder.java
CustomHolder.java:28: incompatible types
found   : java.lang.Object
required: java.lang.String
      str2 = objectHolder.getItem();
                                 ^
1 error
```

This makes sense because the actual type parameter (in this case, `String` or `Object`) dictates the type. When you add a `String` to the `objectHolder`, it is simply stored as an `Object`. When you attempt to assign the `Object` to the `String` (in the call to `objectHolder.getItem`), you now need an explicit cast to the `String` type.

Because of type erasure, it is possible to assign a generic class reference to a reference of its nongeneric (legacy) version. Therefore, the following code compiles without error:

```
Vector oldVector;
Vector<Integer> intVector;

oldVector = intVector; // valid
```

However, though not an error, assigning a reference to a nongeneric class to a reference to a generic class will cause an unchecked compiler warning. This happens when an erasure changes the argument types of a method or a field assignment to a raw type if the erasure changes the method/field type. As an example, the following program causes the warnings shown after it. You must pass `-Xlint:unchecked` on the command line to `javac` to see the specific warnings:

```
import java.util.*;

public class UncheckedExample {
    public void processIntVector(Vector<Integer> v)
    {
        // perform some processing on the vector
    }

    public static void main(String args[])
    {
        Vector<Integer> intVector = new Vector<Integer>();
        Vector oldVector = new Vector();
        UncheckedExample ue = new UncheckedExample();

        // This is permitted
```

```
            oldVector = intVector;
            // This causes an unchecked warning
            intVector = oldVector;
            // This is permitted
            ue.processIntVector(intVector);
            // This causes an unchecked warning
            ue.processIntVector(oldVector);
        }
    }
```

Attempting to compile the previous code leads to the following compiler warnings:

```
UncheckedExample.java:16: warning: unchecked assignment: java.util.Vector to
java.util.Vector<java.lang.Integer>
       intVector = oldVector; // This causes an unchecked warning

UncheckedExample.java:18: warning: unchecked method invocation:
processIntVector(java.util.Vector<java.lang.Integer>) in UncheckedExample is
applied to (java.util.Vector)
        ue.processIntVector(oldVector); // This causes an unchecked warning

2 warnings
```

Wildcards and Bounded Type Variables

Because you can't use `CustomHolder<Object>` as if it were a super-type of `CustomHolder<String>`, you can't write a method that would process both `CustomHolder<Object>` and `CustomHolder<String>`. There is, however, a special way to accomplish this. As part of the generics syntax, a wildcard is introduced, which, when used, basically means "any type parameter." Revisit the previous example and show how the wildcard, a single question mark, is used.

Take the `CustomHolder` class and add a few new methods and a new `main` as follows:

```
    public static void processHolderObject(CustomHolder2<Object> holder)
    {
       Object obj = holder.getItem();

       System.out.println("Item is: " + obj);
    }

    public static void processHolderString(CustomHolder2<String> holder)
    {
       Object obj = holder.getItem();

       System.out.println("Item is: " + obj);
    }

    public static void processHolderWildcard (CustomHolder2<?> holder)
    {
       Object obj = holder.getItem();

       System.out.println("Item is: " + obj);
    }

    public static void main(String args[])
    {
       CustomHolder2<String> stringHolder = new CustomHolder2<String>();
       CustomHolder2<Object> objectHolder = new CustomHolder2<Object>();
       String str = new String("test string");
       String str2;

       stringHolder.putItem(str);
       objectHolder.putItem(str);

       //processHolderObject(stringHolder);
       processHolderObject(objectHolder);

       processHolderString(stringHolder);
       //processHolderString(objectHolder);

       processHolderWildcard(stringHolder);
       processHolderWildcard(objectHolder);
    }
```

The two lines that are commented will prevent the program from compiling. If both these lines are uncommented, the compiler issues the following errors:

```
c:\>javac CustomHolder2.java
CustomHolder2.java:48: processHolderObject(CustomHolder2<java.lang.Object>) in C
ustomHolder2<E> cannot be applied to (CustomHolder2<java.lang.String>)
      processHolderObject(stringHolder);
      ^
CustomHolder2.java:52: processHolderString(CustomHolder2<java.lang.String>) in C
ustomHolder2<E> cannot be applied to (CustomHolder2<java.lang.Object>)
      processHolderString(objectHolder);
      ^
2 errors
```

This reminds you that the type parameter used must match the formal type parameter. However, notice that neither line that invokes `processHolderWildcard` is commented. This is because using the wildcard allows you to pass in either `stringHolder` or `objectHolder`. You can read the method parameter type `CustomerHolder2<?>` as "a CustomHolder2 of any type" as opposed to "of `Object` type" or "of `String` type."

A type parameter can be restricted to certain types by what is called a bound. A bound can be applied to a regular type parameter or a wildcard. Revisit the `Shape` example from earlier in the chapter, which defines a `Shape` interface:

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

interface Shape {
   void draw();
}

class Square implements Shape {
   public void draw()
   {
      System.out.println("Drawing square");
   }
}

class Circle implements Shape {
   public void draw()
   {
      System.out.println("Drawing circle");
   }
}
```

Now define a `PaintProgram` class to demonstrate bounds. If you add a `drawShape` method that defines a type parameter, this won't work:

```
      public static <S> void drawShape(S shape)
      {
         shape.draw();
      }
```

So you must add a bound to the type parameter so Java treats the `shape` formal type parameter as a `Shape` and not an `Object`. By bounding the type parameter to `Shape`, you dictate that the object passed in must derive directly or indirectly from `Shape`. Because of this, Java knows that the object is a `Shape` and thus can invoke methods that belong to `Shape` instead of only `Object` methods:

```
      public static <S extends Shape> void drawShapeBounded(S shape)
      {
         shape.draw();
      }
```

As alluded to earlier, this may make you wonder if generics really are that useful. If you have to explicitly state the bounds on a type parameter, you may as well just use the `Shape` interface to constrain a normal method parameter. One of the places generics really do shine is easing the use of collections, and this is probably the main justification for adding generics to Java.

Look at implementing a `drawAllShapes` method that takes a parameterized `ArrayList`. As expected, you need a bound here so Java does not treat the contents of the `ArrayList` as `Objects`:

```
      public static <T extends Shape> void drawAllShapes(ArrayList<T> shapeList)
      {
         T shape;
         Iterator<T> shapeIterator;

         shapeIterator = shapeList.iterator();
         while(shapeIterator.hasNext()) {
            shape = shapeIterator.next();
            shape.draw();
         }
      }
```

By constraining the T type parameter, invoking draw is acceptable because Java knows it's a `Shape`.

If you want to specify multiple interfaces/classes to use as a bound, separate them with the ampersand (`&`). Also note that `extends` is used to specify bounds regardless of whether the type parameter is bounded by an interface or a class.

Using Generics

It is straightforward to create objects of a generic type. Any parameters must match the bounds specified. Although you might expect to create an array of a generic type, this is only possible with the wildcard type parameter. It is also possible to create a method that works on generic types. This section describes these usage scenarios.

Class Instances

Creating an object of a generic class consists of specifying types for each parameter and supplying any necessary arguments to the constructor. The conditions for any bounds on type variables must be met. Note that only reference types are valid as parameters when creating an instance of a generic class. Trying to use a primitive data type causes the compiler to issue an unexpected type error.

This is a simple creation of a `HashMap` that assigns `Floats` to `Strings`:

```
HashMap<String,Float> hm = new HashMap<String,Float>();
```

Arrays

Arrays of generic types and arrays of type variables are not allowed. Attempting to create an array of parameterized `Vectors`, for example, causes a compiler error:

```
import java.util.*;

public class GenericArrayExample {
    public static void main(String args[])
    {
        Vector<Integer> vectorList[] = new Vector<Integer>[10];
    }
}
```

If you try to compile that code, the compiler issues the following two errors. This code is the simplest approach to creating an array of a generic type and the compiler tells you explicitly that creating a generic type array is forbidden:

```
GenericArrayExample.java:6: arrays of generic types are not allowed
         Vector<Integer> vectorList[] = new Vector<Integer>[10];
                         ^
GenericArrayExample.java:6: arrays of generic types are not allowed
         Vector<Integer> vectorList[] = new Vector<Integer>[10];
                                            ^
2 errors
```

You can, however, create an array of `any type` by using the wildcard as the type parameter.

Generic Methods

In addition to the generic mechanism for classes, generic methods are introduced. The angle brackets for the parameters appear after all method modifiers but before the return type of the method. Following is an example of a declaration of a generic method:

```
static <Elem> void swap(Elem[] a, int i, int j)
{
    Elem temp  = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```

The syntax for the parameters in a generic method is the same as that for generic classes. Type variables can have bounds just like they do in class declarations. Two methods cannot have the same name and argument types. If two methods have the same name and argument types, and have the same number of type variables with the same bounds, then these methods are the same and the compiler will generate an error.

Generics and Exceptions

Type variables are not permitted in `catch` clauses, but can be used in `throws` lists of methods. An example of using a type variable in the `throws` clause follows. The `Executor` interface is designed to execute a section of code that may throw an exception specified as a parameter. In this example, the code that fills in the `execute` method might throw an `IOException`. The specific exception, `IOException`, is specified as a parameter when creating a concrete instance of the `Executor` interface:

```
import java.io.*;

interface Executor<E extends Exception> {
    void execute() throws E;
}

public class GenericExceptionTest {
    public static void main(String args[]) {
        try {
            Executor<IOException> e =
                new Executor<IOException>() {
                public void execute() throws IOException
                {
                    // code here that may throw an
                    // IOException or a subtype of
                    // IOException
                }
            };

            e.execute();
        } catch(IOException ioe) {
            System.out.println("IOException: " + ioe);
            ioe.printStackTrace();
        }
    }
}
```

The specific type of exception is specified when an instance of the `Executor` class is created inside main. The `execute` method throws an arbitrary exception that it is unaware of until a concrete instance of the `Executor` interface is created.

Enhanced for Loop

The `for` loop has been modified to provide a cleaner way to process an iterator. Using a `for` loop with an iterator is error prone because of the slight mangling of the usual form of the `for` loop since the `update` clause is placed in the body of the loop. Some languages have a `foreach` keyword that cleans up the syntax for processing iterators. Java opted not to introduce a new keyword, instead deciding to keep it simple and introduce a new use of the colon. Traditionally, a developer will write the following code to use an iterator:

```
for(Iterator iter = intArray.iterator(); iter.hasNext(); ) {
    Integer intObject = (Integer)iter.next();
    // ... more statements to use intObject ...
}
```

The problem inherent in this code lies in the missing `update` clause of the `for` loop. The code that advances the iterator is moved into the body of the `for` loop out of necessity, because it also returns the next object. The new and improved syntax that does the same thing as the previous code snippet is as follows:

```
for(Integer intObject : intArray) {
    // ... same statements as above go here ...
}
```

This code is much cleaner and easier to read. It eliminates all the potential from the previous construct to introduce errors into the program. If this is coupled with a generic collection, the type of the object is checked versus the type inside the collection at compile time.

Support for this new `for` loop requires a change only to the compiler. The code generated is no different from the same code written in the traditional way. The compiler might translate the previous code into the following, for example:

```
for(Iterator<Integer> $iter = intArray.iterator(); $iter.hasNext(); ) {
    Integer intObject = $iter.next();
    // ... statements ...
}
```

The use of the dollar sign in the identifier in this example merely means the compiler generates a unique identifier for the expansion of the new `for` loop syntax into the more traditional form before compiling.

The same syntax for using an iterator on a collection works for an array. Using the new `for` loop syntax on an array is the same as using it on a collection:

```
for(String strObject : stringArray) {
    // ... statements here using strObject ...
}
```

However, the compiler expands the array version to code slightly longer than the collection version:

```
String[] $strArray = stringArray;

for(int $i = 0; $i < $strArray.length; $i++) {
    String strObject = $strArray[$i];
    // ... statements here ...
}
```

The compiler this time uses two temporary and unique variables during the expansion. The first is an alias to the array, and the second is the loop counter.

Additions to the Java Class Library

To fully support the new `for` loop syntax, the object iterated over must be an array or inherit from a new interface, `java.lang.Iterable`, directly or indirectly. The existing collection classes were retrofitted for the release of JDK 5. The new `Iterable` interface looks like this:

```
public interface Iterable {
    /**
     * Returns an iterator over the elements in this collection.  There are no
     * guarantees concerning the order in which the elements are returned
     * (unless this collection is an instance of some class that provides a
     * guarantee).
     *
     * @return an Iterator over the elements in this collection.
     */
    SimpleIterator iterator();
}
```

Additionally, `java.util.Iterator` will be retrofitted to implement `java.lang.ReadOnlyIterator`, as shown here:

```
public interface ReadOnlyIterator {
    /**
     * Returns true if the iteration has more elements. (In other
     * words, returns true if next would return an element
     * rather than throwing an exception.)
     *
     * @return true if the iterator has more elements.
     */
    boolean hasNext();

    /**
     * Returns the next element in the iteration.
     *
     * @return the next element in the iteration.
     * @exception NoSuchElementException iteration has no more elements.
     */
    Object next();
}
```

The introduction of this interface prevents dependency on the `java.util` interfaces. The change in the `for` loop syntax is at the language level and it makes sense to ensure that any support needed in the class library is located in the `java.lang` branch.

Variable Arguments

C and C++ are a couple of the languages that support variable length argument lists for functions. Java decided to introduce this aspect into the language. Only use variable argument parameter lists in cases that make sense. If you abuse them, it's easy to create source code that is confusing. The C language uses the ellipsis (three periods) in the function declaration to stand for "an arbitrary number of parameters, zero or more." Java also uses the ellipsis but combines it with a type and identifier. The type can be anything — any class, any primitive type, even array types. When using it in an array, however, the ellipsis must come last in the type description, after the square brackets. Because of the nature of variable arguments, each method can only have a single type as a variable argument and it must come last in the parameter list.

Following is an example of a method that takes an arbitrary number of primitive integers and returns their sum:

```
public int sum(int... intList)
{
    int i, sum;

    sum=0;
    for(i=0; i<intList.length; i++) {
        sum += intList[i];
    }

    return(sum);
}
```

All arguments passed in from the position of the argument marked as variable and beyond are combined into an array. This makes it simple to test how many arguments were passed in. All that is needed is to reference the `length` property on the array, and the array also provides easy access to each argument.

Here's a full sample program that adds up all the values in an arbitrary number of arrays:

```
public class VarArgsExample {
    int sumArrays(int[]... intArrays)
    {
        int sum, i, j;

        sum=0;
        for(i=0; i<intArrays.length; i++) {
            for(j=0; j<intArrays[i].length; j++) {
                sum += intArrays[i][j];
            }
        }

        return(sum);
    }

    public static void main(String args[])
    {
        VarArgsExample va = new VarArgsExample();
        int sum=0;

        sum = va.sumArrays(new int[]{1,2,3},
                           new int[]{4,5,6},
                           new int[]{10,16});
        System.out.println("The sum of the numbers is: " + sum);
    }
}
```

This code follows the established approach to defining and using a variable argument. The ellipsis comes after the square brackets (that is, after the variable argument's type). Inside the method the argument `intArrays` is simply an array of arrays.

Boxing and Unboxing Conversions

One tedious aspect of the Java language in the past is the manual operation of converting primitive types (such as `int` and `char`) to their corresponding reference type (for example, `Integer` for `int` and `Character` for `char`). The solution to getting rid of this constant wrapping and unwrapping is boxing and unboxing conversions.

Boxing Conversions

A boxing conversion is an implicit operation that takes a primitive type, such as `int`, and automatically places it inside an instance of its corresponding reference type (in this case, `Integer`). Unboxing is the reverse operation, taking a reference type, such as `Integer`, and converting it to its primitive type, `int`. Without boxing, you might add an `int` primitive to a collection (which holds `Object` types) by doing the following:

```
Integer intObject;
int intPrimitive;
ArrayList arrayList = new ArrayList();

intPrimitive = 11;
intObject = new Integer(intPrimitive);
arrayList.put(intObject); // cannot add intPrimitive directly
```

Although this code is straightforward, it is more verbose than necessary. With the introduction of boxing conversions, the preceding code can be rewritten as follows:

```
int intPrimitive;
ArrayList arrayList = new ArrayList();

intPrimitive = 11;
// here intPrimitive is automatically wrapped in an Integer
arrayList.put(intPrimitive);
```

The need to create an `Integer` object to place an `int` into the collection is no longer needed. The boxing conversion happens such that the resulting reference type's `value()` method (such as `intValue()` for `Integer`) equals the original primitive type's value. Consult the following table for all valid boxing conversions. If there is any other type, the boxing conversion becomes an identity conversion (converting the type to its own type). Note that due to the introduction of boxing conversions, several forbidden conversions referring to primitive types are no longer forbidden because they now can be converted to certain reference types.

Primitive Type	Reference Type
`boolean`	`Boolean`
`byte`	`Byte`
`char`	`Character`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`

Unboxing Conversions

Java also introduces unboxing conversions, which convert a reference type (such as `Integer` or `Float`) to its primitive type (such as `int` or `float`). Consult the following table for a list of all valid unboxing conversions. The conversion happens such that the `value` method of the reference type equals the resulting primitive value.

Reference Type	Primitive Type
`Boolean`	`boolean`
`Byte`	`byte`
`Character`	`char`
`Short`	`short`
`Integer`	`int`
`Long`	`long`
`Float`	`float`
`Double`	`double`

Valid Contexts for Boxing and Unboxing Conversions

Because the boxing and unboxing operations are conversions, they happen automatically with no specific instruction by the programmer (unlike casting, which is an explicit operation). There are several contexts in which boxing and unboxing conversions can happen.

Assignments

An assignment conversion happens when the value of an expression is assigned to a variable. When the type of the expression does not match the type of the variable, and there is no risk of data loss, the conversion happens automatically. The precedence of conversions that happen is the identity conversion, a widening primitive conversion, a widening reference conversion, and then the new boxing (or unboxing) conversion. If none of these conversions are valid, the compiler issues an error.

Method Invocations

When a method call is made, and the argument types don't match precisely with those passed in, several conversions are possible. Collectively, these conversions are known as method invocation conversions. Each parameter that does not match precisely in type to the corresponding parameter in the method signature might be subject to a conversion. The possible conversions are the identity conversion, a widening primitive conversion, a widening reference conversion, and then the new boxing (or unboxing) conversion.

The most specific method must be chosen anytime more than one method matches a particular method call. The rules to match the most specific method change slightly with the addition of boxing conversions. If all the standard checks for resolving method ambiguity fail, the boxing/unboxing conversion won't be used to resolve ambiguity. Therefore, by the time checks are performed for boxing conversions, the method invocation is deemed ambiguous and fails.

Combining boxing with generics allows you to write the following code:

```
import java.util.*;

public class BoxingGenericsExample {
    public static void main(String args[])
    {
        HashMap<String,Integer> hm = new HashMap<String,Integer>();

        hm.put("speed", 20);
    }
}
```

The primitive integer `20` is automatically converted to an `Integer` and then placed into the `HashMap` under the specified key.

Static Imports

Importing static data is introduced into the language to simplify using static attributes and methods. After importing static information, the methods/attributes can then be used without the need to qualify the method or attribute with its class name. For example, by importing the static members of the `Math` class, you can write `abs` or `sqrt` instead of `Math.abs` and `Math.sqrt`.

This mechanism also prevents the dangerous coding practice of placing a set of static attributes into an interface, and then in each class that needs to use the attributes, implementing that interface. The following interface should not be implemented in order to use the attributes without qualification:

```
interface ShapeNumbers {
    public static int CIRCLE = 0;
    public static int SQUARE = 1;
    public static int TRIANGLE = 2;
}
```

Implementing this interface creates an unnecessary dependence on the `ShapeNumbers` interface. Even worse, it becomes awkward to maintain as the class evolves, especially if other classes need access to these constants also and implement this interface. It is easy for compiled classes to get out of synchronization with each other if the interface containing these attributes changes and only some classes are recompiled.

To make this cleaner, the static members are placed into a class (instead of an interface) and then imported via a modified syntax of the import directive. `ShapeNumbers` is revised to the following:

```
package MyConstants;

class ShapeNumbers {
    public static int CIRCLE = 0;
    public static int SQUARE = 1;
    public static int TRIANGLE = 2;
}
```

A client class then imports the static information from the `ShapeNumbers` class and can then use the attributes `CIRCLE`, `SQUARE`, and `TRIANGLE` without the need to prefix them with `ShapeNumbers` and the member operator.

To import the static members in your class, specify the following in the import section of your Java source file (at the top):

```
import static MyConstants.ShapeNumbers.*; // imports all static data
```

This syntax is only slightly modified from the standard format of the import statement. The keyword `static` is added after the `import` keyword, and instead of importing packages, you now always add on the class name because the static information is being imported from a specific class. The chief reason the keyword `static` is added to the import statement is to make it clear to those reading the source code that the import is for the static information.

You can also import constants individually by using the following syntax:

```
import static MyConstants.ShapeNumbers.CIRCLE;
import static MyConstants.ShapeNumbers.SQUARE;
```

This syntax is also what you would expect. The keyword `static` is included because this is a static import, and the pieces of static information to import are each specified explicitly.

You cannot statically import data from a class that is inside the default package. The class must be located inside a named package. Also, static attributes and methods can conflict. For example, following are two classes (located in `Colors.java` and `Fruits.java`) containing static constants:

```
package MyConstants;

public class Colors {
    public static int white = 0;
    public static int black = 1;
    public static int red = 2;
    public static int blue = 3;
    public static int green = 4;
    public static int orange = 5;
    public static int grey = 6;
}
```

```
package MyConstants;

public class Fruits {
    public static int apple = 500;
    public static int pear = 501;
    public static int orange = 502;
    public static int banana = 503;
    public static int strawberry = 504;
}
```

If you write a class that tries to statically import data on both these classes, everything is fine until you try to use a static variable that is defined in both of them:

```
import static MyConstants.Colors.*;
import static MyConstants.Fruits.*;

public class StaticTest {
    public static void main(String args[])
    {
        System.out.println("orange = " + orange);
        System.out.println("color orange = " + Colors.orange);
        System.out.println("Fruity orange = " + Fruits.orange);
    }
}
```

The seventh line of the program causes the following compiler error. The identifier `orange` is defined in both `Colors` and `Fruits`, so the compiler cannot resolve this ambiguity:

```
StaticTest.java:7: reference to orange is ambiguous, both variable orange in
MyConstants.Colors and variable orange in MyConstants.Fruits match
         System.out.println("orange = " + orange);
```

In this case, you should explicitly qualify the conflicting name with the class where it is defined. Instead of writing `orange`, write `Colors.orange` or `Fruits.orange`.

Enumerations

Java introduces enumeration support at the language level in the JDK 5 release. An enumeration is an ordered list of items wrapped into a single entity. An instance of an enumeration can take on the value of any single item in the enumeration's list of items. The simplest possible enumeration is the `Colors enum` shown here:

```
public enum Colors { red, green, blue }
```

They present the ability to compare one arbitrary item to another, and to iterate over the list of defined items. An enumeration (abbreviated `enum` in Java) is a special type of class. All enumerations implicitly subclass a new class in Java, `java.lang.Enum`. This class cannot be subclassed manually.

There are many benefits to built-in support for enumerations in Java. Enumerations are type-safe and the performance is competitive with constants. The constant names inside the enumeration don't need to be qualified with the enumeration's name. Clients aren't built with knowledge of the constants inside the enumeration, so changing the enumeration is easy without having to change the client. If constants are removed from the enumeration, the clients will fail and you'll receive an error message. The names of the constants in the enumeration can be printed, so you get more information than simply the ordinal number of the item in the list. This also means that the constants can be used as names for collections such as `HashMap`.

Because an enumeration is a class in Java, it can also have fields and methods, and implement interfaces. Enumerations can be used inside `switch` statements in a straightforward manner, and are relatively simple for programmers to understand and use.

Here's a basic `enum` declaration and its usage inside a `switch` statement. If you want to track what operating system a certain user is using, you can use an enumeration of operating systems, which are defined in the `OperatingSystems enum`. Note that because an enumeration is effectively a class, it cannot be public if it is in the same file as another class that is public. Also note that in the `switch` statement, the constant names cannot be qualified with the name of the enumeration they are in. The details are automatically handled by the compiler based on the type of the `enum` used in the `switch` clause:

```
import java.util.*;

enum OperatingSystems {
    windows, unix, linux, macintosh
}

public class EnumExample1 {
    public static void main(String args[])
    {
        OperatingSystems os;

        os = OperatingSystems.windows;
        switch(os) {
            case windows:
                System.out.println("You chose Windows!");
                break;
            case unix:
                System.out.println("You chose Unix!");
                break;
            case linux:
                System.out.println("You chose Linux!");
                break;
            case macintosh:
                System.out.println("You chose Macintosh!");
                break;
            default:
                System.out.println("I don't know your OS.");
                break;
        }
    }
}
```

The `java.lang.Enum` class implements the `Comparable` and `Serializable` interfaces. The details of comparing enumerations and serializing them to a data source are already handled inside the class. You cannot mark an `enum` as `abstract` unless every constant has a class body, and these class bodies override the abstract methods in the `enum`. Also note that enumerations cannot be instantiated using `new`. The compiler will let you know that `enum types may not be instantiated`.

Java introduces two new collections, `EnumSet` and `EnumMap`, which are only meant to optimize the performance of sets and maps when using `enums`. Enumerations can be used with the existing collection classes, or with the new collections when optimization tailored to enumerations is desired.

Methods can be declared inside an `enum`. There are restrictions placed on defining constructors, however. Constructors can't chain to superclass constructors, unless the superclass is another `enum`. Each constant inside the `enum` can have a class body, but because this is effectively an anonymous class, you cannot define a constructor.

You can also add attributes to the enumeration and to the individual `enum` constants. An `enum` constant can also be followed by arguments, which are passed to the constructor defined in the `enum`.

Here's an example enumeration with fields and methods:

```
enum ProgramFlags {
    showErrors(0x01),
    includeFileOutput(0x02),
    useAlternateProcessor(0x04);

    private int bit;

    ProgramFlags(int bitNumber)
    {
        bit = bitNumber;
    }

    public int getBitNumber()
    {
        return(bit);
    }
}

public class EnumBitmapExample {
    public static void main(String args[])
```

```
        {
            ProgramFlags flag = ProgramFlags.showErrors;

            System.out.println("Flag selected is: " +
                                    flag.ordinal() +
                               " which is " +
                                    flag.name());
        }
    }
```

The `ordinal()` method returns the position of the constant in the list. The value of `showErrors` is 0 because it comes first in the list, and the ordinal values are 0-based. The `name()` method can be used to get the name of the constant, which provides for getting more information about enumerations.

Metadata

Another feature that Sun has decided to include in the JDK 5 release of Java is a metadata facility. This enables tagging classes with extra information that tools can analyze, and also applying certain blocks of code to classes automatically. The metadata facility is introduced in the `java.lang.annotation` package. An annotation is the association of a `tag` to a construct in Java such as a class, known as a *target* in annotation terminology. The types of constructs that can be annotated are listed in the `java.lang.annotation.ElementType` enumeration, and are listed in the following table. Even annotations can be annotated. `TYPE` covers classes, interfaces, and `enum` declarations.

ElementType Constant
`ANNOTATION_TYPE`
`CONSTRUCTOR`
`FIELD`
`LOCAL_VARIABLE`
`METHOD`
`PACKAGE`
`PARAMETER`
`TYPE`

Another concept introduced is the life of an annotation, known as the *retention*. Certain annotations may only be useful at the Java source code level, such as an annotation for the `javadoc` tool. Others might be needed while the program is executing. The `RetentionPolicy` enumeration lists three `type` lifetimes for an annotation. The `SOURCE` policy indicates the annotations should be discarded by the compiler, that is, should only available at the source code level. The `CLASS` policy indicates that the annotation should appear in the class file, but is possibly discarded at runtime. The `RUNTIME` policy indicates the annotations should make it through to the executing program, and these can then be viewed using reflection.

Several types of annotations are defined in this package. These are listed in the following table. Each of these annotations inherits from the `Annotation` interface, which defines an `equals` method and a `toString` method.

Annotation Class Name	Description
`Target`	Specifies to which program elements an annotation type is applicable. Each program element can appear only once.
`Documented`	Specifies annotations should be documented by `javadoc` or other documentation tools. This can only be applied to annotations.
`Inherited`	Inherits annotations from superclasses, but not interfaces. The policy on this annotation is `RUNTIME`, and it can be applied only to annotations.
`Retention`	Indicates how long annotations on this program element should be available. See `RetentionPolicy` discussed previously. The policy on this annotation is `RUNTIME`, and it can be applied only to annotations.
`Deprecated`	Marks a program element as deprecated, telling developers they should no longer use it. Retention policy is `SOURCE`.
`Overrides`	Indicates that a method is meant to override the method in a parent class. If the override does not actually exist, the compiler will generate an error message. This can be applied only to methods.

Two useful source level annotations come with JDK 5, `@deprecated` and `@overrides`. The `@deprecated` annotation is used to mark a method as deprecated — that is, it shouldn't be used by client programmers. The compiler will issue a warning when encountering this annotation on a class method that a programmer uses. The other annotation, `@overrides`, is used to mark a method as overriding a method in the parent class. The compiler will ensure that a method marked as `@overrides` does indeed override a method in the parent class. If the method in the child class doesn't override the one in the parent class, the compiler will issue an error alerting the programmer to the fact that the method signature does not match the method in the parent class.

Developing a custom annotation isn't difficult. Create a `CodeTag` annotation that stores basic author and modification date information, and also stores any bug fixes applied to that piece of code. The annotation will be limited to classes and methods:

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.SOURCE)
@Target({ElementType.TYPE, ElementType.METHOD})
public @interface CodeTag {
    String authorName();
    String lastModificationDate();
    String bugFixes() default "";
}
```

The Retention is set to SOURCE, which means this annotation is not available during compile time and runtime. The doclet API is used to access source level annotations. The Target is set to TYPE (for classes/interfaces/enums) and METHOD for methods. A compiler error is generated if the CodeTag annotation is applied to any other source code element. The first two annotation elements are authorName and lastModificationDate, both of which are mandatory. The bugFixes element defaults to the empty string if not specified. Following is an example class that utilizes the CodeTag annotation:

```
import java.lang.annotation.*;

@CodeTag(authorName="Dilbert",
         lastModificationDate="May 7, 2006")
public class ServerCommandProcessor {
    @CodeTag(authorName="Dilbert",
             lastModificationDate="May 10, 2006",
             bugFixes="BUG0170")
    public void setParams(String serverName)
    {
        // ...
    }

    public void executeCommand(String command, Object... params)
    {
        // ...
    }
}
```

Note how annotation is used to mark who modified the source and when. The method was last modified a day after the class because of the bug fix. This custom annotation can be used to track this information as part of keeping up with source code modifications. To view or process these source code annotations, the doclet API must be used.

The doclet API (aka Javadoc API) has been extended to support the processing of annotations in the source code. To use the doclet API, include the tools.jar file (located in lib directory of a default JDK install, version 5 or higher) in your classpath. You use the doclet API by writing a Java class that extends com.sun.javadoc.Doclet. The start method must be implemented because this is the method that Javadoc invokes on a doclet to perform custom processing. A simple doclet to print out all classes and methods in a Java source file follows:

```
import com.sun.javadoc.*;

public class ListClasses extends Doclet {
    public static boolean start(RootDoc root) {
        ClassDoc[] classes = root.classes();
        for (ClassDoc cd : classes) {
            System.out.println("Class [" + cd + "] has the following methods");
            for(MemberDoc md : cd.methods()) {
               System.out.println("  " + md);
            }
        }
        return true;
    }
}
```

The `start` method takes a `RootDoc` as a parameter, which is automatically passed in by the `javadoc` tool. The `RootDoc` provides the starting point to obtain access to all elements inside the source code, and also information on the command line such as additional packages and classes.

The interfaces added to the doclet API for annotations are `AnnotationDesc`, `AnnotationDesc.ElementValuePair`, `AnnotationTypeDoc`, `AnnotationTypeElementDoc`, and `AnnotationValue`.

Any element of Java source that can have annotations has an `annotations()` method associated with the doclet API's counterpart to the source code element. These are `AnnotationTypeDoc`, `AnnotationTypeElementDoc`, `ClassDoc`, `ConstructorDoc`, `ExecutableMemberDoc`, `FieldDoc`, `MethodDoc`, and `MemberDoc`. The `annotations()` method returns an array of `AnnotationDesc`.

AnnotationDesc

This class represents an annotation, which is an annotation type (`AnnotationTypeDoc`), and an array of annotation type elements paired with their values. `AnnotationDesc` defines the following methods.

Method	**Description**
`AnnotationTypeDoc annotationType()`	Returns this annotation's type.
`AnnotationDesc.ElementValuePair[] elementValues()`	Returns an array of an annotation's elements and their values. Only elements explicitly listed are returned. The elements that aren't listed explicitly, which assume their default value, are not returned because this method processes just what is listed. If there are no elements, an empty array is returned.

AnnotationDesc.ElementValuePair

This represents an association between an annotation type's element and its value. The following methods are defined.

Method	**Description**
`AnnotationTypeElementDoc element()`	Returns the annotation type element.
`AnnotationValue value()`	Returns the annotation type element's value.

AnnotationTypeDoc

This interface represents an annotation in the source code, just like `ClassDoc` represents a `Class`. Only one method is defined.

Method	**Description**
`AnnotationTypeElementDoc[] elements()`	Returns an array of the elements of this annotation type.

AnnotationTypeElementDoc

This interface represents an element of an annotation type.

Method	Description
`AnnotationValue defaultValue()`	Returns the default value associated with this annotation type, or null if there is no default value.

AnnotationValue

This interface represents the value of an annotation type element.

Method	Description
`String toString()`	Returns a string representation of the value.
`Object value()`	Returns the value. The object behind this value could be any of the following. ❑ A wrapper class for a primitive type (such as `Integer` or `Float`) ❑ A `String` ❑ A `Type` (representing a class, a generic class, a type variable, a wildcard type, or a primitive data type) ❑ A `FieldDoc` (representing an `enum` constant) ❑ An `AnnotationDesc` ❑ An array of `AnnotationValue`

Here's an example using the annotation support provided by the doclet API. This doclet echoes all annotations and their values that it finds in a source file:

```
import com.sun.javadoc.*;
import java.lang.annotation.*;

public class AnnotationViewer {
    public static boolean start(RootDoc root)
    {
        ClassDoc[] classes = root.classes();

        for (ClassDoc cls : classes) {
            showAnnotations(cls);
        }

        return(true);
    }

    static void showAnnotations(ClassDoc cls)
    {
        System.out.println("Annotations for class [" + cls + "]");
```

```
            process(cls.annotations());

            System.out.println();
            for(MethodDoc m : cls.methods()) {
                System.out.println("Annotations for method [" + m + "]");
                process(m.annotations());
                System.out.println();
            }
        }

        static void process(AnnotationDesc[] anns)
        {
            for (AnnotationDesc ad : anns) {
                AnnotationDesc.ElementValuePair evp[] = ad.elementValues();

                for(AnnotationDesc.ElementValuePair e : evp) {
                    System.out.println("  NAME: " + e.element() +
                                       ", VALUE=" + e.value());
                }

            }
        }
    }
```

The `start` method iterates across all classes (and interfaces) found in the source file. Because all annotations on source code elements are associated with the `AnnotationDesc` interface, a single method can be written to process annotations regardless of with which source code element the annotation is associated. The `showAnnotations` method prints out annotations associated with the current class and then processes all methods inside that class. The doclet API makes processing these source code elements easy. To execute the doclet, pass the name of the doclet and name of the class to process on the command line as follows:

```
javadoc -doclet AnnotationViewer ServerCommandProcessor.java
```

The doclet echoes the following to the screen:

```
Loading source file ServerCommandProcessor.java...
Constructing Javadoc information...
Annotations for class [ServerCommandProcessor]
  NAME: CodeTag.authorName(), VALUE="Dilbert"
  NAME: CodeTag.lastModificationDate(), VALUE="May 7, 2006"

Annotations for method [ServerCommandProcessor.setParams(java.lang.String)]
  NAME: CodeTag.authorName(), VALUE="Dilbert"
  NAME: CodeTag.lastModificationDate(), VALUE="May 10, 2006"
  NAME: CodeTag.bugFixes(), VALUE="BUG0170"

Annotations for method [ServerCommandProcessor.executeCommand(java.lang.String,
java.lang.Object[])]
```

To access annotations at runtime, the reflection API must be used. This support is built in through the interface `AnnotatedElement`, which is implemented by the reflection classes `AccessibleObject`, `Class`, `Constructor`, `Field`, `Method`, and `Package`. All these elements may have annotations. The `AnnotatedElement` interface defines the following methods.

Method	Description
`<T extends Annotation> T getAnnotation(Class<T> annotationType)`	Returns the annotation associated with the specified type, or null if none exists.
`Annotation[] getAnnotations()`	Returns an array of all annotations on the current element, or a zero-length array if no annotations are present.
`Annotation[] getDeclaredAnnotations()`	Similar to `getAnnotations` but does not return inherited annotations — only annotations explicitly declared on this element are returned. Returns a zero-length array if no annotations are present.
`boolean isAnnotationPresent(Class<? extends Annotation> annotationType)`	Returns `true` if the `annotationType` is present on the current element, `false` otherwise.

Develop an annotation that might be useful in developing a testing framework. The framework invokes test methods specified in the annotation and expects a Boolean return value from these testing methods. The reflection API is used to both process the annotation and execute the test methods.

The annotation is listed as follows:

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
public @interface TestParameters {
    String testStage();
    String testMethods();
    String testOutputType(); // "db" or "file"
    String testOutput(); // filename or data source/table name
}
```

An example application of this annotation is to a class of utility methods for strings. You might develop your own utility class and develop testing methods to ensure the utility methods work:

```
@TestParameters(testStage="Unit",
                testMethods="testConcat,testSubstring",
                testOutputType="screen",
                testOutput="")
public class StringUtility {
    public String concat(String s1, String s2)
    {
        return(s1 + s2);
    }

    public String substring(String str, int startIndex, int endIndex)
    {
        return(str.substring(startIndex, endIndex));
    }

    public boolean testConcat()
    {
        String s1 = "test";
        String s2 = " 123";

        return(concat(s1,s2).equals("test 123"));
    }

    public boolean testSubstring()
    {
        String str = "The cat landed on its feet";

        return(substring(str, 4, 7).equals("cat"));
    }
}
```

Following is an example implementation of the testing framework. It uses reflection to process the annotation and then invoke the testing methods, writing the results to the screen (though other output destinations can be built into the framework).

```
    import java.lang.reflect.*;
    import java.lang.annotation.*;
    import java.util.*;

    public class TestFramework {
        static void executeTests(String className) {
            try {
                Object obj = Class.forName(className).newInstance();

                TestParameters tp = obj.getClass().getAnnotation(TestParameters.class);
                if(tp != null) {
                    String methodList = tp.testMethods();
                    StringTokenizer st = new StringTokenizer(methodList, ",");
                    while(st.hasMoreTokens()) {
                        String methodName = st.nextToken().trim();

                        Method m = obj.getClass().getDeclaredMethod(methodName);
                        System.out.println("");
                        System.out.println(methodName);
                        System.out.println("----------------");
                        String result = invoke(m, obj);
                        System.out.println("Result: " + result);
                    }
                } else {
                    System.out.println("No annotation found for " + obj.getClass());
                }
            } catch(Exception ex) {
                System.out.println("Exception: " + ex);
                ex.printStackTrace();
            }
        }

        static String invoke(Method m, Object o) {
            String result = "PASSED";

            try {
                m.invoke(o);
            } catch(Exception ex) {
                System.out.println("Exception: " + ex + "\n" + ex.getCause());
                result = "FAILED";
            }

            return(result);
        }

        public static void main(String [] args) {
            if(args.length == 0) {
               System.out.println("Must specify class name (without an extension)");
            } else {
               executeTests(args[0]);
            }
        }
    }
```

Executing the preceding class on the `StringUtility` class provides the following output:

```
    C:\>java TestFramework StringUtility

    testConcat
    ----------------
    Result: PASSED

    testSubstring
    ----------------
    Result: PASSED
```

The `executeTests` method obtains a handle to the `TestParameters` annotation from the class and then invokes each method from the `testMethods()` element of the annotation. This is a simple implementation of the testing framework and can be extended to support the other elements of the `TestParameters` annotation, such as writing results to a database instead of the screen. This is a practical example of using metadata — adding declarative information to Java source that can then be utilized by external programs and/or doclets for generating documentation.

Important Java Utility Libraries

This section describes several key utility libraries in Java. These libraries are as follows:

❑ **Java logging:** A powerful logging system that is vital for providing meaningful error messages to end users, developers, and people working in the field.

❑ **Regular expressions:** A powerful "miniature language" used to process strings in a variety of ways, such as searching for substrings that match a particular pattern.

❑ **Java preferences:** A way to store and retrieve both system- and user-defined configuration options.

Each library is designed for flexibility of usage. Familiarity with these libraries is vital when developing solutions in Java. The more tools on your belt as a developer, the better equipped you are.

Java Logging

Java has a well-designed set of classes to control, format, and publish messages through the logging system. It is important for a program to log error and status messages. There are many people who can benefit from logging messages, including developers, testers, end users, and people working in the field that have to troubleshoot programs without source code. It is vital to include a high number of quality log messages in a program, from status updates to error conditions (such as when certain exceptions are caught). By using the logging system, it is possible to see what the program is doing without consulting the source code, and most importantly, track down error conditions to a specific part of the program. The value of a logging system is obvious, especially in large systems where a casual error with minimal or no log messages might take days or longer to track down.

The logging system in `java.util.logging` is sophisticated, including a way to prioritize log messages such that only messages a particular logger is interested in get logged, and the messages can be output to any source that a `Handler` object can handle. Examples of logging destinations are files, databases, and output streams. Take a close look at Figure 1-1 to see an overview of the entire logging system.

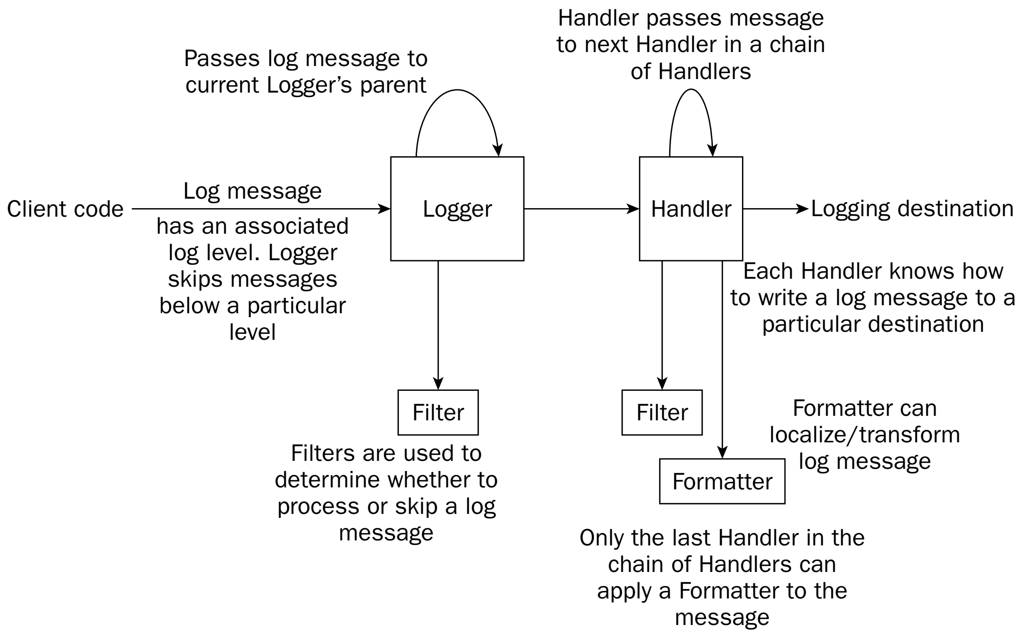
Passes log message to current Logger's parent
Handler passes message to next Handler in a chain of Handlers
Client code
Log message
has an associated log level. Logger skips messages below a particular level
Logger
Handler
Logging destination
Each Handler knows how to write a log message to a particular destination
Filter
Filter
Formatter can localize/transform log message
Formatter
Filters are used to determine whether to process or skip a log message
Only the last Handler in the chain of Handlers can apply a Formatter to the message

Figure 1-1

The specific `Logger` objects are actually hierarchical, and though not mandatory, can mirror the class hierarchy. When a `Logger` receives a log message, the message is also passed automatically to the parent of `Logger`. The root logger is named `" "` (the empty string) and has no parent. Each other `Logger` is usually named something such as `java.util` or `java.util.ArrayList` to mirror the package/class hierarchy. The names of the Logger objects, going down the tree, are dot-separated. Therefore, java .util is the parent Logger of java.util.ArrayList. You can name the loggers any arbitrary string, but keeping with the dot-separated convention helps with clarity.

The simplest use of the logging system creates a Logger and uses all system defaults (defined in a properties file) for the logging system. The following example outputs the log message using a formatting class called the SimpleFormatter that adds time/date/source information to the log message:

```
import java.util.logging.*;

public class BasicLoggingExample {
    public static void main(String args[])
    {
        Logger logger = Logger.getLogger("BasicLoggingExample");

        logger.log(Level.INFO, "Test of logging system");
    }
}
```

The following is output from the BasicLoggingExample:

```
Feb 22, 2004 4:07:06 PM BasicLoggingExample main
INFO: Test of logging system
```

The Log Manager

The entire logging system for a particular application is controlled by a single instance of the LogManager class. This instance is created during the initialization of the LogManager. The LogManager contains the hierarchical namespace that has all the named Logger objects. The LogManager also contains logging control properties that are used by Handlers and other objects in the logging system for configuration. These configuration properties are stored in the file lib/logging.properties that is located in the JRE installation path.

There are two system properties that can be used to initialize the logging system with different properties. The first way is to override the property java.util.logging.config.file and specify the full path to your own version of logging.properties. The other property, java.util.logging.config .class, is used to point to your own LogManager. This custom LogManager is responsible for reading in its configuration. If neither of these properties is set, Java will default to the logging.properties file in the JRE directory. Consult the following table for properties that can be set on the LogManager in this file. You can also specify properties for Loggers and Handlers in this file. These properties are described later in this section.

Property Key	Property Value
Handlers	Comma-separated list of Handler classes. Each handler must be located somewhere in the system classpath.
.level	Sets the minimum level for a specific Logger. The level must be prefixed with the full path to a specific Logger. A period by itself sets the level for the root logger.

The LogManager Class

The `LogManager` class contains methods to configure the current instance of the logging system through a number of configuration methods, tracks loggers and provides access to these loggers, and handles certain logging events. These methods are listed in the following tables.

Configuration

The methods listed in the following table relate to storage and retrieval of configuration information in the `LogManager`.

Method	**Description**
`String getProperty(String name)`	Returns the value corresponding to a specified logging property.
`void readConfiguration()`	Reloads the configuration using the same process as startup. If the system properties controlling initialization have not changed, the same file that was read at startup will be read here.
`void readConfiguration(InputStream ins)`	Reads configuration information from an `InputStream` that is in the `java.util.Properties` format.
`void reset()`	Resets the logging system. All `Handlers` are closed and removed and all logger levels except on the root are set to null. The root logger's level is set to `Level.INFO`.

Logger Control

The methods listed in the following table relate to the storage, retrieval, and management of individual `Logger` references. These are the most commonly used methods on the `LogManager` class.

Method	**Description**
`static LogManager getLogManager()`	Returns the one and only instance of the `LogManager` object.
`boolean addLogger(Logger logger)`	Returns true if the `Logger` passed in is not already registered (its name isn't already in the list). The logger is registered. Returns false if the name of the `Logger` object already exists in the list of registered loggers.
`Logger getLogger(String name)`	Returns a reference to the `Logger` object that is named "name," or null if no logger is found.
`Enumeration getLoggerNames()`	Returns an `Enumeration` containing a list of the names of all currently registered loggers.

Events

The methods listed in the following table provide a way to add and remove references to listeners that should be notified when properties are changed on the `LogManager`.

Method	Description
`void addPropertyChangeListener (PropertyChangeListener l)`	Adds a property change listener to the list of listeners that want notification of when a property has changed. The same listener can be added multiple times.
`void removePropertyChangeListener (PropertyChangeListener l)`	Removes a single occurrence of a property change listener in the list of listeners.

The Logger Class

An instance of the `Logger` class is used by client code to log a message. Both the log message and each logger have an associated level. If the level of the log message is equal to or greater than the level of the logger, the message is then processed. Otherwise, the logger drops the log message. It is an inexpensive operation to test whether or not to drop the log message, and this operation is done at the entry point to the logging system — the `Logger` class. These levels are defined inside the `Level` class. Consult the following table for a full list of levels.

Logger Level	Description
`SEVERE`	Highest logging level. This has top priority.
`WARNING`	One level below severe. Intended for warning messages that need attention, but aren't serious.
`INFO`	Two levels below severe. Intended for informational messages.
`CONFIG`	Three levels below severe. Intended for configuration-related output.
`FINE`	Four levels below severe. Intended for program tracing information.
`FINER`	Five levels below severe. Intended for program tracing information.
`FINEST`	Lowest logging level. This has lowest priority.
`ALL`	Special level that makes the system log ALL messages.
`OFF`	Special level that makes the system log NO messages (turns logging off completely).

Logger Methods

The `Logger` is the main class used in code that utilizes the logging system. Methods are provided to obtain a named or anonymous logger, configure and get information about the logger, and log messages.

Obtaining a Logger

The following methods allow you to retrieve a handle to a `Logger`. These are static methods and provide an easy way to obtain a `Logger` without going through a `LogManager`.

Method	Description

`static Logger getAnonymousLogger()`
`static Logger getAnonymousLogger(String resourceBundleName)`
	Creates an anonymous logger that is exempt from standard security checks, for use in applets. The anonymous logger is not registered in the `LogManager` namespace, but has the root logger (“”) as a parent, inheriting level and handlers from the root logger. A resource bundle can also be specified for localization of log messages.

`static Logger getLogger(String name)`
`static Logger getLogger(String name, String resourceBundleName)`
	Returns a named logger from the `LogManager` namespace, or if one is not found, creates and returns a new named logger. A resource bundle can also be specified for localization of log messages.

Configuring a Logger Object

The following methods allow you to configure a `Logger` object. You can add and remove handlers, set the logging level on this `Logger` object, set its parent, and choose whether or not log messages should be passed up the logger hierarchy.

Method	Description
`void addHandler(Handler handler)`	Adds a `Handler` to the logger. Multiple handlers can be added. Also note that the root logger is configured with a set of default `Handlers`.
`void removeHandler(Handler handler)`	Removes a specified handler from the list of handlers on this logger. If the handler is not found, this method returns silently.
`void setLevel(Level newLevel)`	Sets the log level that this logger will use. Message levels lower than the logger’s value will be automatically discarded. If null is passed in, the level will be inherited from this logger’s parent.

Table continued on following page

Method	Description
`void setParent(Logger parent)`	Sets the parent for this logger. This should not be called by application code, because it is intended for use only by the logging system.
`void setUseParentHandlers(boolean useParentHandlers)`	Specifies true if log messages should be passed to their parent loggers, or false to prevent the log messages from passing to their parent.
`Filter getFilter()`	Returns the filter for this logger, which might be null if no filter is associated.
`Handler[] getHandlers()`	Returns an array of all handlers associated with this logger.
`Level getLevel()`	Returns the log level assigned to this logger. If null is returned, it indicates the logging level of the parent logger that will be used.
`String getName()`	Returns the name of this logger, or null if this is an anonymous logger.
`Logger getParent()`	The nearest parent to the current logger is returned, or null if the current logger is the root logger.
`ResourceBundle getResourceBundle()`	Returns the `ResourceBundle` associated with this logger. Resource bundles are used for localization of log messages. If null is returned, the resource bundle from the logger's parent will be used.
`String getResourceBundleName()`	Returns the name of the resource bundle this logger uses for localization, or null if the resource bundle is inherited from the logger's parent.
`boolean getUseParentHandlers()`	Returns true if log messages are passed to the logger's parent, or false if log messages are not passed up the hierarchy.

Logging Messages

The following methods are all used to actually log a message using a `Logger`. Convenience methods are provided for logging messages at each logging level, and also for entering and exiting methods and throwing exceptions. Additional methods are provided to localize log messages using a resource bundle.

Method	Description

`void config(String msg)`
`void fine(String msg)`
`void finer(String msg)`
`void finest(String msg)`
`void info(String msg)`
`void severe(String msg)`
`void warning(String msg)`
	The `Logger` class contains a number of convenience methods for logging messages. For quickly logging a message of a specified level, one method for each logging level is defined.

`void entering(String sourceClass, String sourceMethod)`
`void entering(String sourceClass, String sourceMethod, Object param1)`
`void entering(String sourceClass, String sourceMethod, Object params[])`
	Log a message when a method is first entered. The variant forms take a parameter to the method, or an array of parameters, to provide for more detailed tracking of the method invocation. The message of the log is `ENTRY` in addition to the other information about the method call. The log level is `Level.FINER`.

`void exiting(String sourceClass, String sourceMethod)`
`void exiting(String sourceClass, String sourceMethod, Object result)`
	Log a message when a method is about to return. The log message contains `RETURN` and the log level is `Level.FINER`. The source class and source method are also logged.

`boolean isLoggable(Level level)`	Checks if a certain level will be logged. Returns true if it will be logged, or false otherwise.

`void log(Level level, String msg)`
`void log(Level level, String msg, Object param1)`
`void log(Level level, String msg, Object[] params)`
`void log(Level level, String msg, Throwable thrown)`
`void log(LogRecord record)`
	Standard general logging convenience methods. Variants include the ability to specify a parameter or array of parameters to log, or `Throwable` information. The information is placed into a `LogRecord` object and sent into the logging system. The last variant takes a `LogRecord` object.

Table continued on following page

Method	Description
`void logp(Level level, String sourceClass, String sourceMethod, String msg)`	Take source class and source method names in addition to the other information. All this is put into a `LogRecord` object and sent into the system.
`void logp(Level level, String sourceClass, String sourceMethod, String msg, Object param1)`	
`void logp(Level level, String sourceClass, String sourceMethod, String msg, Object[] params)`	
`void logp(Level level, String sourceClass, String sourceMethod, String msg, Throwable thrown)`	
`void logrb(Level level, String sourceClass, String sourceMethod, String bundleName, String msg)`	Allow you to specify a resource bundle in addition to the other information. The resource bundle will be used to localize the log message.
`void logrb(Level level, String sourceClass, String sourceMethod, String bundleName, String msg, Object param1)`	
`void logrb(Level level, String sourceClass, String sourceMethod, String bundleName, String msg, Object[] params)`	
`void logrb(Level level, String sourceClass, String sourceMethod, String bundleName, String msg, Throwable thrown)`	
`void throwing(String sourceClass, String sourceMethod, Throwable thrown)`	Logs a throwing message. The log level is `Level.FINER`. The log record's message is set to `THROW` and the contents of `thrown` are put into the log record's `thrown` property instead of inside the log record's message.

The LogRecord Class

The `LogRecord` class encapsulates a log message, carrying the message through the logging system. `Handlers` and `Formatters` use `LogRecords` to have more information about the message (such as the time it was sent and the logging level) for processing. If a client to the logging system has a reference to a `LogRecord` object, the object should no longer be used after it is passed into the logging system.

LogRecord Methods

The `LogRecord` contains a number of methods to examine and manipulate properties on a log record, such as message origination, the log record's level, when it was sent into the system, and any related resource bundles.

Method	Description
`Level getLevel()`	Returns the log record's level.
`String getMessage()`	Returns the unformatted version of the log message, before formatting/localization.
`long getMillis()`	Returns the time the log record was created in milliseconds.
`Object[] getParameters()`	Returns an array of parameters of the log record, or null if no parameters are set.
`long getSequenceNumber()`	Returns the sequence number of the log record. The sequence number is assigned in the log record's constructor to create a unique number for each log record.
`Throwable getThrown()`	Returns the `Throwable` associated with this log record, such as the `Exception` if an exception is being logged. Returns null if no `Throwable` is set.
`String getLoggerName()`	Returns the name of the logger, which might be null if it is the anonymous logger.
`String getSourceClassName()`	Gets the name of the class that might have logged the message. This information may be specified explicitly, or inferred from the stack trace and therefore might be inaccurate.
`String getSourceMethodName()`	Gets the name of the method that might have logged the message. This information may be specified explicitly, or inferred from the stack trace and therefore might be inaccurate.
`int getThreadID`	Returns the identifier for the thread that originated the log message. This is an ID inside the Java VM.

Setting Information about Message Origination

The following methods allow you to set origination information on the log message such as an associated exception, class and method that logged the message, and the ID of the originating thread.

Method	Description
`void setSourceClassName (String sourceClassName)`	Sets the name of the class where the log message is originating.
`void setSourceMethodName (String sourceMethodName)`	Sets the name of the method where the log message is originating.

Table continued on following page

Method	Description
`void setThreadID (int threadID)`	Sets the identifier of the thread where the log message is originating.
`void setThrown (Throwable thrown)`	Sets a `Throwable` to associate with the log message. Can be null.

Resource Bundle Methods

The following methods allow you to retrieve and configure a resource bundle for use with the log message. Resource bundles are used for localizing log messages.

Method	Description
`ResourceBundle getResourceBundle()`	Returns the `ResourceBundle` associated with the logger that is used to localize log messages. Might be null if there is no associated `ResourceBundle`.
`String getResourceBundleName()`	Returns the name of the resource bundle used to localize log messages. Returns null if log messages are not localizable (no resource bundle defined).
`void setResourceBundle (ResourceBundle bundle)`	Sets a resource bundle to use to localize log messages.
`void setResourceBundleName(String name)`	Sets the name of a resource bundle to use to localize log messages.

Setting Information about the Message

The following methods configure the log message itself. Some of the information you can configure related to the log message are its level, the contents of the message, and the time the message was sent.

Method	Description
`void setLevel(Level level)`	Sets the level of the logging message.
`void setLoggerName(String name)`	Sets the name of the logger issuing this message. Can be null.
`void setMessage(String message)`	Sets the contents of the message before formatting/localization.
`void setMillis(long millis)`	Sets the time of the log message, in milliseconds, since 1970.
`void setParameters(Object[] parameters)`	Sets parameters for the log message.

Method	Description
`void setSequenceNumber(long seq)`	Sets the sequence number of the log message. This method shouldn't usually be called, because the constructor assigns a unique number to each log message.

The Level Class

The `Level` class defines the entire set of logging levels, and also objects of this class represent a specific logging level that is then used by loggers, handlers, and so on. If you desire, you can subclass this class and define your own custom levels, as long as they do not conflict with the existing logging levels.

Logging Levels

The following logging levels are defined in the `Level` class.

Log Level	Description
`OFF`	Special value that is initialized to `Integer.MAX_VALUE`. This turns logging off.
`SEVERE`	Meant for serious failures. Initialized to 1,000.
`WARNING`	Meant to indicate potential problems. Initialized to 900.
`INFO`	General information. Initialized to 800.
`CONFIG`	Meant for messages useful for debugging. Initialized to 700.
`FINE`	Meant for least verbose tracing information. Initialized to 500.
`FINER`	More detailed tracing information. Initialized to 400.
`FINEST`	Most detailed level of tracing information. Initialized to 300.
`ALL`	Special value. Logs `ALL` messages. Initialized to `Integer.MIN_VALUE`.

Level Methods

The `Level` class defines methods to set and retrieve a specific logging level. Both numeric and textual versions of levels can be used.

Method	Description
`static Level parse(String name)`	Returns a `Level` object representing the name of the level that is passed in. The string `name` can be one of the logging levels, such as `SEVERE` or `CONFIG`. An arbitrary number, between `Integer.MIN_VALUE` and `Integer.MAX_VALUE` can also be passed in (as a string). If the number represents one of the existing level values, that level is returned. Otherwise, a new `Level` is returned corresponding to the passed in value. Any invalid name or number causes an `IllegalArgumentException` to get thrown. If the name is null, a `NullPointerException` is thrown.
`boolean equals(Object ox)`	Returns true if the object passed in has the same level as the current class.
`String getLocalizedName()`	Returns the localized version of the current level's name, or the non-localized version if no localization is available.
`String getName()`	Returns the non-localized version of the current level's name.
`String getResourceBundleName()`	Returns the name of the level's localization resource bundle, or null if no localization resource bundle is defined.
`int hashCode()`	Returns a hash code based on the level value.
`int intValue()`	Returns the integer value for the current level.
`String toString()`	Returns the non-localized name of the current level.

The Handler Class

The `Handler` class is used to receive log messages and then publish them to an external destination. This might be memory, a file, a database, a TCP/IP stream, or any number of places that can store log messages. Just like loggers, a handler has an associated level. Log messages that are less than the level on the handler are discarded. Each specific instance of a `Handler` has its own properties and is usually configured in the `logging.properties` file. The next section discusses the various handlers that are found in the `java.util.logging` package. Creating a custom handler is straightforward, because implementations of only `close()`, `flush()`, and `publish(LogRecord record)` are needed.

Handler Methods

The `Handler` class defines three abstract methods that need specific behavior in inheriting classes. The other methods available on the `Handler` class are for dealing with message encoding, filters, formatters, and error handlers.

Key Abstract Methods

When developing a custom handler, there are three abstract methods that must be overridden. These are listed in the following table.

Method	Description
`abstract void close()`	Should perform a `flush()` and then free any resources used by the handler. After `close()` is called, the `Handler` should no longer be used.
`abstract void flush()`	Flushes any buffered output to ensure it is saved to the associated resource.
`abstract void publish(LogRecord record)`	Takes a log message forwarded by a logger and then writes it to the associated resource. The message should be formatted (using the `Formatter`) and localized.

Set and Retrieve Information about the Handler

The methods listed in the following table allow you to retrieve information about the handler, such as its encoding, associated error manager, filter, formatter, and level, and also set this configuration information.

Method	Description
`String getEncoding()`	Returns the name of the character encoding. If the name is null, the default encoding should be used.
`ErrorManager getErrorManager()`	Returns the `ErrorManager` associated with this `Handler`.
`Filter getFilter()`	Returns the `Filter` associated with this `Handler`, which might be null.
`Formatter getFormatter()`	Returns the `Formatter` associated with this `Handler`, which might be null.
`Level getLevel()`	Returns the level of this handler. Log messages lower than this level are discarded.
`boolean isLoggable(LogRecord record)`	Returns true if the `LogRecord` passed in will be logged by this handler. The checks include comparing the record's level to the handler's, testing against the filter (if one is defined), and any other checks defined in the handler.
`void setEncoding(String encoding)`	Sets the encoding to a specified character encoding. If null is passed in, the default platform encoding is used.
`void setErrorManager (ErrorManager em)`	Sets an `ErrorManager` for the handler. If any errors occur while processing, the `Error Manager's error` method is invoked.
`void setFilter (Filter newFilter)`	Sets a custom filter that decides whether to discard or keep a log message when the `publish` method is invoked.
`void setFormatter (Formatter newFormatter)`	Sets a `Formatter` that performs custom formatting on log messages passed to the handler before the log message is written to the destination.
`void setLevel(Level newLevel)`	Sets the level threshold for the handler. Log messages below this level are automatically discarded.

Stock Handlers

The `java.util.logging` package includes a number of predefined handlers to write log messages to common destinations. These classes include the `ConsoleHandler`, `FileHandler`, `MemoryHandler`, `SocketHandler`, and `StreamHandler`. These classes provide a specific implementation of the abstract methods in the `Handler` class. All the property key names in the tables are prefixed with `java.util.logging` in the actual properties file.

The `StreamHandler` serves chiefly as a base class for all handlers that write log messages to some `OutputStream`. The subclasses of `StreamHandler` are `ConsoleHandler`, `FileHandler`, and `SocketHandler`. A lot of the stream handling code is built into this class. See the following table for a list of properties for the `StreamHandler`.

Property Name	Description	Default Value
`StreamHandler.level`	Log level for the handler	`Level.INFO`
`StreamHandler.filter`	Filter to use	undefined
`StreamHandler.formatter`	Formatter to use	`java.util.logging .SimpleFormatter`
`StreamHandler.encoding`	Character set encoding to use	Default platform encoding

The following methods are defined/implemented on the `StreamHandler` class.

Method	Description
`void close()`	The `head` string from the `Formatter` will be written if it hasn't been already, and the `tail` string is written before the stream is closed.
`void flush()`	Writes any buffered output to the stream (flushes the stream).
`boolean isLoggable (LogRecord record)`	Performs standard checks against `level` and `filter`, but also returns false if no output stream is open or the record passed in is null.
`void publish(LogRecord record)`	If the record passed in is loggable, the `Formatter` is then invoked to format the log message and then the message is written to the output stream.
`void setEncoding(String encoding)`	Sets the character encoding to use for log messages. Pass in null to use the current platform's default character encoding.
`protected void setOutputStream (OutputStream out)`	Sets an `OutputStream` to use. If an `OutputStream` is already open, it is flushed and then closed. The new `OutputStream` is then opened.

The `ConsoleHandler` writes log messages to `System.err`. It subclasses `StreamHandler` but overrides `close()` to only perform a flush, so the `System.err` stream does not get closed. The default formatter used is `SimpleFormatter`. The following table describes the properties that can be defined in the `logging.properties` file for the `ConsoleHandler`.

Property Name	Description	Default Value
`ConsoleHandler.level`	Log level for the handler	`Level.INFO`
`ConsoleHandler.filter`	Filter to use	Undefined
`ConsoleHandler.formatter`	Formatter to use	`java.util.logging .SimpleFormatter`
`ConsoleHandler.encoding`	Character set encoding to use	Default platform encoding

The `SocketHandler` writes log messages to the network over a specified TCP port. The properties listed in the following table are used by the `SocketHandler`. The default constructor uses the properties defined, and a second constructor allows the specification of the host and port `SocketHandler(String host, int port)`. The `close()` method flushes and closes the output stream, and the `publish()` method flushes the stream after each record is written.

Property Name	Description	Default Value
`SocketHandler.level`	Log level for the handler	`Level.INFO`
`SocketHandler.filter`	Filter to use	undefined
`SocketHandler.formatter`	Formatter to use	`java.util.logging .XMLFormatter`
`SocketHandler.encoding`	Character set encoding to use	Default platform encoding
`SocketHandler.host`	Target host name to connect to	undefined
`SocketHandler.port`	Target TCP port to use	undefined

The `FileHandler` is able to write to a single file, or write to a rotating set of files as each file reaches a specified maximum size. The next number in a sequence is added to the end of the name of each rotating file, unless a *generation* (sequence) pattern is specified elsewhere. The properties for the `FileHandler` are listed in the following table.

Property Name	Description	Default Value
`FileHandler.level`	Log level for the handler.	`Level.INFO`
`FileHandler.filter`	Filter to use.	undefined
`FileHandler.formatter`	Formatter to use.	`java.util.logging .XMLFormatter`
`FileHandler.encoding`	Character set encoding to use.	Default platform encoding
`FileHandler.limit`	Specifies approximate maximum number of bytes to write to a file. 0 means no limit.	0

Table continued on following page

Property Name	Description	Default Value
`FileHandler.count`	Specifies how many output files to cycle through.	1
`FileHandler.pattern`	Pattern used to generate output filenames.	`%h/java%u.log`
`FileHandler.append`	Boolean value specifying whether to append to an existing file or overwrite it.	false

The `FileHandler` class supports filename patterns, allowing the substitution of paths such as the user's home directory or the system's temporary directory. The forward slash (/) is used as a directory separator, and this works for both Unix and Windows machines. Also supported is the ability to specify where the generation number goes in the filename when log files are rotated. These patterns are each prefixed with a percent sign (%). To include the percent sign in the filename, specify two percent signs (%%). The following table contains all the valid percent-sign substitutions.

Pattern	Description
`%t`	Full path of the system temporary directory
`%h`	Value of the `user.home` system property
`%g`	Generation number used to distinguish rotated logs
`%u`	Unique number used to resolve process conflicts

For example, if you're executing this on Windows 95 and specify the filename pattern `%t/app_log.txt`, the `FileHandler` class expands this to `C:\TEMP\app_log.txt`. Note that the `%t` and `%h` commands do not include the trailing forward slash.

The `%u` is used to account for when multiple threads/processes will access the same log file. Only one process can have the file open for writing, so to prevent the loss of logging information, the `%u` can be used to output to a log file that has a similar name to the others. For example, the filename pattern `%t/logfile%u.txt` can be specified, and if two processes open this same log file for output, the first will open `C:\TEMP\logfile0.txt` and the second will open `C:\TEMP\logfile1.txt`.

The `MemoryHandler` is a circular buffer in memory. It is intended for use as a quick way to store messages, so the messages have to be sent to another handler to write them to an external source. Because the buffer is circular, older log records eventually are overwritten by newer records. Formatting can be delayed to another `Handler`, which makes logging to a `MemoryHandler` quick. Conditions that will cause the `MemoryHandler` to send data (push data) to another `Handler` are as follows:

- ❑ A log record passed in has a level greater than a specified `pushLevel`.
- ❑ Another class calls the `push` method on the `MemoryHandler`.
- ❑ A subclass implements specialized behavior to push data depending on custom criteria.

The properties on the `MemoryHandler` are listed in the following table.

Property Name	Description	Default Value
`MemoryHandler.level`	Log level for the handler	`Level.INFO`
`MemoryHandler.filter`	Filter to use	undefined
`MemoryHandler.size`	Size of the circular buffer (in bytes)	1,000
`MemoryHandler.push`	Defines the push level — the minimum level that will cause messages to be sent to the target handler	`Level.SEVERE`
`MemoryHandler.target`	Specifies the name of the target `Handler` class	(undefined)

The constructors create a `MemoryHandler` with a default or specific configuration.

Constructor	Description
`MemoryHandler()`	Creates a `MemoryHandler` based on the configuration properties.
`MemoryHandler(Handler target, int size, Level pushLevel)`	Creates a `MemoryHandler` with a specified target handler, size of the buffer, and push level.

The methods provided by the `MemoryHandler` create and configure the behavior of the memory handler.

Method	Description
`void publish(LogRecord record)`	Stores the record in the internal buffer, if it is loggable (see `isLoggable`). If the level of the log record is greater than or equal to the `pushLevel`, all buffered records, including the current one, are written to the target `Handler`.
`void close()`	Closes the handler and frees the associated resources. Also invokes `close` on the target handler.
`void flush()`	Causes a `flush`, which is different from a `push`. To actually write the log records to a destination other than memory, a `push` must be performed.
`Level getPushLevel()`	Returns the current push level.
`boolean isLoggable (LogRecord record)`	Compares the log levels, and then runs the record through the filter if one is defined. Whether or not the record will cause a `push` is ignored by this method.
`void push()`	Sends all records in the current buffer to the target handler, and clears the buffer.
`void setPushLevel(Level newLevel)`	Sets a new `push` level.

The Formatter Class

The `Formatter` class is used to perform some custom processing on a log record. This formatting might be localization, adding additional program information (such as adding the time and date to log records), or any other processing needed. The `Formatter` returns a string that is the processed log record. The `Formatter` class also has support for `head` and `tail` strings that come before and after all log records. An example that will be implemented later in this section is a custom `Formatter` that writes log records to an HTML table. For this formatter, the head string would be the `<table>` tag, and the tail string is the `</table>` tag. The methods defined in the `Formatter` class are listed in the following table.

Method	Description
`abstract String format(LogRecord record)`	Performs specific formatting of the log record and returns the formatted string.
`String formatMessage(LogRecord record)`	The message string in the `LogRecord` is localized using the record's `ResourceBundle`, and formatted according to `java.text` style formatting (replacing strings such as `{0}`).
`String getHead(Handler h)`	Returns the header string for a specified handler, which can be null.
`String getTail(Handler h)`	Returns the tail string for a specified handler, which can be null.

Stock Formatters

The logging package comes already equipped with a couple of useful formatters. The `SimpleFormatter` provides a basic implementation of a formatter. The `XMLFormatter` outputs log records in a predefined XML format. These two stock formatters will cover a variety of basic logging scenarios, but if you need behavior not supplied by either of these formatters, you can write your own.

SimpleFormatter

The `SimpleFormatter` does a minimal level of work to format log messages. The format method of the `SimpleFormatter` returns a one- or two-line summary of the log record that is passed in. Logging a simple log message, such as `test 1`, using the `SimpleFormatter` will issue the following output:

```
Apr 18, 2004 12:18:25 PM LoggingTest main
INFO: test 1
```

The `SimpleFormatter` formats the message with the date, time, originating class name, originating method name, and on the second line, the level of the log message and the log message itself.

XMLFormatter

The `XMLFormatter` formats the log records according to an XML DTD. You can use the `XMLFormatter` with any character encoding, but it is suggested that it be used only with `"UTF-8"`. The `getHead()` and getTail() methods are used to output the start and end of the XML file, the parts that aren't repeated for each log record but are necessary to create a valid XML file.

Example output from the XMLFormatter follows:

```
<?xml version="1.0" encoding="windows-1252" standalone="no"?>
<!DOCTYPE log SYSTEM "logger.dtd">
<log>
<record>
  <date>2004-04-18T12:22:36</date>
  <millis>1082305356235</millis>
  <sequence>0</sequence>
  <logger>LoggingTest</logger>
  <level>INFO</level>
  <class>LoggingTest</class>
  <method>main</method>
  <thread>10</thread>
  <message>test 1</message>
</record>
<record>
  <date>2004-04-18T12:22:36</date>
  <millis>1082305356265</millis>
  <sequence>1</sequence>
  <logger>LoggingTest</logger>
  <level>INFO</level>
  <class>LoggingTest</class>
  <method>main</method>
  <thread>10</thread>
  <message>test 2</message>
</record>
</log>
```

The XML DTD that the logging system uses is shown here:

```
<!-- DTD used by the java.util.logging.XMLFormatter -->
<!-- This provides an XML formatted log message. -->

<!-- The document type is "log" which consists of a sequence
of record elements -->
<!ELEMENT log (record*)>

<!-- Each logging call is described by a record element. -->
<!ELEMENT record (date, millis, sequence, logger?, level,
class?, method?, thread?, message, key?, catalog?, param*, exception?)>

<!-- Date and time when LogRecord was created in ISO 8601 format -->
<!ELEMENT date (#PCDATA)>

<!-- Time when LogRecord was created in milliseconds since
midnight January 1st, 1970, UTC. -->
<!ELEMENT millis (#PCDATA)>

<!-- Unique sequence number within source VM. -->
```

```
<!ELEMENT sequence (#PCDATA)>

<!-- Name of source Logger object. -->
<!ELEMENT logger (#PCDATA)>

<!-- Logging level, may be either one of the constant
names from java.util.logging.Constants (such as "SEVERE"
or "WARNING") or an integer value such as "20". -->
<!ELEMENT level (#PCDATA)>

<!-- Fully qualified name of class that issued
logging call, e.g. "javax.marsupial.Wombat". -->
<!ELEMENT class (#PCDATA)>

<!-- Name of method that issued logging call.
It may be either an unqualified method name such as
"fred" or it may include argument type information
in parenthesis, for example "fred(int,String)". -->
<!ELEMENT method (#PCDATA)>

<!-- Integer thread ID. -->
<!ELEMENT thread (#PCDATA)>

<!-- The message element contains the text string of a log message. -->
<!ELEMENT message (#PCDATA)>

<!-- If the message string was localized, the key element provides
the original localization message key. -->
<!ELEMENT key (#PCDATA)>

<!-- If the message string was localized, the catalog element provides
the logger's localization resource bundle name. -->
<!ELEMENT catalog (#PCDATA)>

<!-- If the message string was localized, each of the param elements
provides the String value (obtained using Object.toString())
of the corresponding LogRecord parameter. -->
<!ELEMENT param (#PCDATA)>

<!-- An exception consists of an optional message string followed
by a series of StackFrames. Exception elements are used
for Java exceptions and other java Throwables. -->
<!ELEMENT exception (message?, frame+)>

<!-- A frame describes one line in a Throwable backtrace. -->
<!ELEMENT frame (class, method, line?)>

<!-- an integer line number within a class's source file. -->
<!ELEMENT line (#PCDATA)>
```

Creating Your Own Formatter

It isn't too difficult to develop a custom `Formatter`. As an example, here's an implementation of the `HTMLTableFormatter` that was mentioned previously. The HTML code that is output looks like this:

```
<table border>
    <tr><th>Time</th><th>Log Message</th></tr>
    <tr><td>...</td><td>...</td></tr>
    <tr><td>...</td><td>...</td></tr>
</table>
```

Each log record starts with `<tr>` and ends with `</tr>` because there is only one log record per table row. The `<table>` tag and the first row of the table make up the `head` string. The `</table>` tag makes up the tail of the collection of log records. The custom formatter only needs an implementation of the `getHead()`, `getTail()`, and `format(LogRecord record)` methods:

```
import java.util.logging.*;

class HTMLTableFormatter extends java.util.logging.Formatter {
    public String format(LogRecord record)
    {
        return("  <tr><td>" +
               record.getMillis() +
               "</td><td>" +
               record.getMessage() +
               "</td></tr>\n");
    }

    public String getHead(Handler h)
    {
        return("<table border>\n  " +
               "<tr><th>Time</th><th>Log Message</th></tr>\n");
    }

    public String getTail(Handler h)
    {
        return("</table>\n");
    }
}
```

The Filter Interface

A filter is used to provide additional criteria to decide whether to discard or keep a log record. Each logger and each handler can have a filter defined. The `Filter` interface defines a single method:

```
boolean isLoggable(LogRecord record)
```

The `isLoggable` method returns true if the log message should be published and false if it should be discarded.

Creating Your Own Filter

An example of a custom filter is a filter that discards any log message that does not start with "client". This is useful if log messages are coming from a number of sources, and each log message from a particular client (or clients) is prefixed with the string "client":

```
import java.util.logging.*;

public class ClientFilter implements java.util.logging.Filter {
    public boolean isLoggable(LogRecord record)
    {
        if(record.getMessage().startsWith("client"))
            return(true);
        else
            return(false);
    }
}
```

The ErrorManager

The ErrorManager is associated with a handler and is used to handle any errors that occur, such as exceptions that are thrown. The client of the logger most likely does not care or cannot handle errors, so using an ErrorManager is a flexible and straightforward way for a Handler to report error conditions. The error manager defines a single method:

```
void error(String msg, Exception ex, int code)
```

This method takes the error message (a string), the Exception thrown, and a code representing what error occurred. The codes are defined as static integers in the ErrorManager class and are listed in the following table.

Error Code	Description
CLOSE_FAILURE	Used when close() fails
FLUSH_FAILURE	Used when flush() fails
FORMAT_FAILURE	Used when formatting fails for any reason
GENERIC_FAILURE	Used for any other error that other error codes don't match
OPEN_FAILURE	Used when open of an output source fails
WRITE_FAILURE	Used when writing to the output source fails

Logging Examples

By default, log messages are passed up the hierarchy to each parent. Following is a small program that uses a named logger to log a message using the XMLFormatter:

```
import java.util.logging.*;

public class LoggingExample1 {
    public static void main(String args[])
```

```
        {
            try{
                LogManager lm = LogManager.getLogManager();
                Logger logger;
                FileHandler fh = new FileHandler("log_test.txt");

                logger = Logger.getLogger("LoggingExample1");

                lm.addLogger(logger);
                logger.setLevel(Level.INFO);
                fh.setFormatter(new XMLFormatter());

                logger.addHandler(fh);
                // root logger defaults to SimpleFormatter.
                // We don't want messages logged twice.
                //logger.setUseParentHandlers(false);
                logger.log(Level.INFO, "test 1");
                logger.log(Level.INFO, "test 2");
                logger.log(Level.INFO, "test 3");
                fh.close();
            } catch(Exception e) {
                System.out.println("Exception thrown: " + e);
                e.printStackTrace();
            }
        }
    }
```

What happens here is the XML output is sent to `log_test.txt`. This file is as follows:

```
<?xml version="1.0" encoding="windows-1252" standalone="no"?>
<!DOCTYPE log SYSTEM "logger.dtd">
<log>
<record>
  <date>2004-04-20T2:09:55</date>
  <millis>1082472395876</millis>
  <sequence>0</sequence>
  <logger>LoggingExample1</logger>
  <level>INFO</level>
  <class>LoggingExample1</class>
  <method>main</method>
  <thread>10</thread>
  <message>test 1</message>
</record>
<record>
  <date>2004-04-20T2:09:56</date>
  <millis>1082472396096</millis>
  <sequence>1</sequence>
  <logger>LoggingExample1</logger>
  <level>INFO</level>
  <class>LoggingExample1</class>
  <method>main</method>
  <thread>10</thread>
  <message>test 2</message>
</record>
</log>
```

Because the log messages are then sent to the parent logger, the messages are also output to System.err using the SimpleFormatter. The following is output:

```
Feb 11, 2004 2:09:55 PM LoggingExample1 main
INFO: test 1
Feb 11, 2004 2:09:56 PM LoggingExample1 main
INFO: test 2
```

Here's a more detailed example that uses the already developed HTMLTableFormatter. Two loggers are defined in a parent-child relationship, ParentLogger and ChildLogger. The parent logger will use the XMLFormatter to output to a text file, and the child logger will output using the HTMLTableFormatter to a different file. By default, the root logger will execute and the log messages will go to the console using the SimpleFormatter. The HTMLTableFormatter is extended to an HTMLFormatter to generate a full HTML file (instead of just the table tags):

```
import java.util.logging.*;
import java.util.*;

class HTMLFormatter extends java.util.logging.Formatter {
    public String format(LogRecord record)
    {
        return("      <tr><td>" +
               (new Date(record.getMillis())).toString() +
               "</td>" +
               "<td>" +
               record.getMessage() +
               "</td></tr>\n");
    }

    public String getHead(Handler h)
    {
        return("<html>\n  <body>\n" +
               "    <table border>\n      " +
               "<tr><th>Time</th><th>Log Message</th></tr>\n");
    }

    public String getTail(Handler h)
    {
        return("    </table>\n  </body>\n</html>");
    }
}

public class LoggingExample2 {
    public static void main(String args[])
    {
        try {
            LogManager lm = LogManager.getLogManager();
            Logger parentLogger, childLogger;
            FileHandler xml_handler = new FileHandler("log_output.xml");
            FileHandler html_handler = new FileHandler("log_output.html");
            parentLogger = Logger.getLogger("ParentLogger");
            childLogger = Logger.getLogger("ParentLogger.ChildLogger");

            lm.addLogger(parentLogger);
            lm.addLogger(childLogger);

            // log all messages, WARNING and above
            parentLogger.setLevel(Level.WARNING);
            // log ALL messages
            childLogger.setLevel(Level.ALL);
            xml_handler.setFormatter(new XMLFormatter());
            html_handler.setFormatter(new HTMLFormatter());

            parentLogger.addHandler(xml_handler);
            childLogger.addHandler(html_handler);

            childLogger.log(Level.FINE, "This is a fine log message");
            childLogger.log(Level.SEVERE, "This is a severe log message");
            xml_handler.close();
            html_handler.close();
        } catch(Exception e) {
            System.out.println("Exception thrown: " + e);
            e.printStackTrace();
        }
    }
}
```

Here's what gets output to the screen:

```
Apr 20, 2004 12:43:09 PM LoggingExample2 main
SEVERE: This is a severe log message
```

Here's what gets output to the `log_output.xml` file:

```
<?xml version="1.0" encoding="windows-1252" standalone="no"?>
<!DOCTYPE log SYSTEM "logger.dtd">
<log>
<record>
  <date>2004-04-20T12:43:09</date>
  <millis>1082479389122</millis>
  <sequence>0</sequence>
  <logger>ParentLogger.ChildLogger</logger>
  <level>FINE</level>
  <class>LoggingExample2</class>
  <method>main</method>
  <thread>10</thread>
  <message>This is a fine log message</message>
</record>
<record>
  <date>2004-04-20T12:43:09</date>
  <millis>1082479389242</millis>
  <sequence>1</sequence>
  <logger>ParentLogger.ChildLogger</logger>
  <level>SEVERE</level>
  <class>LoggingExample2</class>
  <method>main</method>
  <thread>10</thread>
  <message>This is a severe log message</message>
</record>
</log>
```

The contents of the `log_output.html` file are as follows:

```
<html>
  <body>
    <table border>
      <tr><th>Time</th><th>Log Message</th></tr>
      <tr><td>Tue Apr 20 12:43:09 EDT 2004</td><td>This is a fine log
message</td></tr>
      <tr><td>Tue Apr 20 12:43:09 EDT 2004</td><td>This is a severe log
message</td></tr>
    </table>
  </body>
</html>
```

Note that the root logger, by default, logs messages at level `INFO` and above. However, because the `ParentLogger` is only interested in levels at `WARNING` and above, log messages with lower levels are immediately discarded. The HTML file contains all log messages because the `ChildLogger` is set to process all log messages. The XML file contains only the one `SEVERE` log message, because log messages below the `WARNING` level are discarded.

Regular Expressions

Regular expressions are a powerful facility available to solve problems relating to the searching, isolating, and/or replacing of chunks of text inside strings. The subject of regular expressions (sometimes abbreviated regexp or regexps) is large enough that it deserves its own book — and indeed, books have been devoted to regular expressions. This section provides an overview of regular expressions and discusses the support Sun has built in to the `java.util.regex` package.

Regular expressions alleviate a lot of the tedium of working with a simple parser, providing complex pattern matching capabilities. Regular expressions can be used to process text of any sort. For more sophisticated examples of regular expressions, consult another book that is dedicated to regular expressions.

If you've never seen regular expressions before in a language, you've most likely seen a small subset of regular expressions with file masks on Unix/DOS/Windows. For example, you might see the following files in a directory:

```
Test.java
Test.class
StringProcessor.java
StringProcessor.class
Token.java
Token.class
```

You can type `dir *.*` at the command line (on DOS/Windows) and every file will be matched and listed. The asterisks are replaced with any string, and the period is taken literally. If the file mask `T*.class` is used, only two files will be matched — `Test.class` and `Token.class`. The asterisks are considered meta-characters, and the period and letters are considered normal characters. The meta-characters are part of the regular expression "language," and Java has a rich set of these that go well beyond the simple support in file masks. The normal characters match literally against the string being tested. There is also a facility to interpret meta-characters literally in the regular expression language.

Several examples of using regular expressions are examined throughout this section. As an initial example, assume you want to generate a list of all classes inside Java files that have no modifier before the keyword `class`. Assuming you only need to examine a single line of source code, all you have to do is ignore any white space before the string `class`, and you can generate the list.

A traditional approach would need to find the first occurrence of `class` in a string and then ensure there's nothing but white space before it. Using regular expressions, this task becomes much easier. The entire Java regular expression language is examined shortly, but the regular expression needed for this case is `\s*class`. The backslash is used to specify a meta-character, and in this case, `\s` matches any white space. The asterisk is another meta-character, standing for "0 or more occurrences of the previous term." The word `class` is then taken literally, so the pattern stands for matching white space (if any exists) and then matching `class`. The Java code to use this pattern is shown next:

```
Pattern pattern = Pattern.compile("\\s*class");
// Need two backslashes to preserve the backslash

Matcher matcher = pattern.matcher("\t\t    class");
if(matcher.matches()) {
    System.out.println("The pattern matches the string");
} else {
    System.out.println("The pattern does not match the string");
}
```

This example takes a regular expression (stored in a `Pattern` object) and uses a matcher to see if the regular expression matches a specific string. This is the simplest use of the regular expression routines in Java. Consult Figure 1-2 for an overview of how the regular expression classes work with each other.

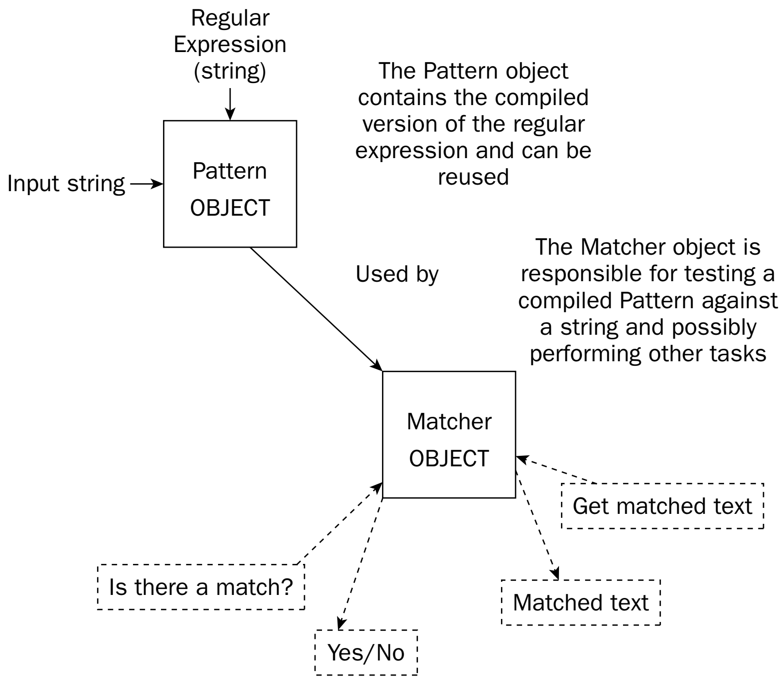
Regular Expression (string)
The Pattern object contains the compiled version of the regular expression and can be reused
Input string
Pattern OBJECT
Used by
The Matcher object is responsible for testing a compiled Pattern against a string and possibly performing other tasks
Matcher OBJECT
Get matched text
Is there a match?
Matched text
Yes/No

Figure 1-2

The designers of the regular expression library decided to use a *Pattern-Matcher* model, which separates the regular expression from the matcher itself. The regular expression is compiled into a more optimized form by the `Pattern` class. This compiled pattern can then be used with multiple matchers, or reused by the same matcher matching on different strings.

In a regular expression, any single character matches literally, except for just a few exceptions. One such exception is the period (.), which matches any single character in the string that is being analyzed. There are sets of meta-characters predefined to match specific characters. These are listed in the following table.

Meta-Character	Matches
`\\`	A single backslash
`\0n`	An octal value describing a character, where `n` is a number such that 0 <= n <= 7
`\0nn`	
`\0mnn`	An octal value describing a character, where `m` is 0 <= m <= 3 and `n` is 0 <= n <= 7
`\0xhh`	The character with hexadecimal value `hh` (where 0 <= h <= F)
`\uhhhh`	The character with hexadecimal value `hhhh` (where 0 <= h <= F)
`\t`	A tab (character '`\u0009`')
`\n`	A newline (linefeed) ('`\u000A`')
`\r`	A carriage-return ('`\u000D`')
`\f`	A form-feed ('`\u000C`')
`\a`	A bell/beep character ('`\u0007`')
`\e`	An escape character ('`\u001B`')
`\cx`	The control character corresponding to `x`, such as `\cc` is control-c
`.`	Any single character

The regular expression language also has meta-characters to match against certain string boundaries. Some of these boundaries are the beginning and end of a line, and the beginning and end of words. The full list of boundary meta-characters can be seen in the following table.

Meta-Character	Matches
`^`	Beginning of the line
`$`	End of the line
`\b`	A word boundary
`\B`	A non-word boundary

Meta-Character	Matches
`\A`	The beginning of the input
`\G`	The end of the previous match
`\Z`	The end of the input before any line terminators (such as carriage-return or linefeed)
`\z`	The end of the input

Regular expression languages also have character classes, which are a way of specifying a list of possible characters that can match any single character in the string you want to match. If you want to specify a character class explicitly, the characters go between square brackets. Therefore, the character class `[0123456789]` matches any single digit. It is also possible to specify "any character except one of these" by using the caret after the first square bracket. Using the expression `[^012]`, any single digit *except* for 0, 1, and 2 is matched. You can specify character ranges using the dash. The character class `[a-z]` matches any single lowercase letter, and `[^a-z]` matches any character except a lowercase letter. Any character range can be used, such as `[0-9]` to match a single digit, or `[0-3]` to match a 0, 1, 2, or 3. Multiple ranges can be specified, such as `[a-zA-Z]` to match any single letter. The regular expression package contains a set of predefined character classes, and these are listed in the following table.

Character Class Meta-Character	Matches
`.`	Any single character
`\d`	A digit `[0-9]`
`\D`	A nondigit `[^0-9]`
`\s`	A whitespace character `[ \t\n\x0B\f\r]`
`\S`	A nonwhitespace character `[^\s]`
`\w`	A word character `[a-zA-Z_0-9]`
`\W`	A nonword character `[^\w]`

Additionally, there are POSIX character classes and Java character classes. These are listed in the following tables, respectively.

Character Class Meta-Character	Matches
`\p{Lower}`	Lowercase letter `[a-z]`
`\p{Upper}`	Uppercase letter `[A-Z]`
`\p{ASCII}`	All ASCII `[\x00-\x7F]`

Table continued on following page

Character Class Meta-Character	Matches
`\p{Alpha}`	Any lowercase or uppercase letter
`\p{Digit}`	A digit `[0-9]`
`\p{Alnum}`	Any letter or digit
`\p{Punct}`	Punctuation ``[!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~]``
`\p{Graph}`	A visible character: any letter, digit, or punctuation
`\p{Print}`	A printable character; same as `\p{Graph}`
`\p{Blank}`	A space or tab `[ \t]`
`\p{Cntrl}`	A control character `[\x00-\x1F\x7F]`
`\p{XDigit}`	Hexadecimal digit `[0-9a-fA-F]`
`\p{Space}`	A whitespace character `[ \t\n\x0B\f\r]`

Character Class	Matches
`\p{javaLowerCase}`	Everything that `Character.isLowerCase()` matches
`\p{javaUpperCase}`	Everything that `Character.isUpperCase()` matches
`\p{javaWhitespace}`	Everything that `Character.isWhitespace()` matches
`\p{javaMirrored}`	Everything that `Character.isMirrored()` matches

Another feature of the regular expression language is the ability to match a particular character a specified number of times. In the previous example, the asterisk was used to match zero or more characters of white space. There are two general ways the repetition operators work. One class of operators is greedy, that is, they match as much as they can, until the end. The other class is reluctant (or lazy), and matches only to the first chance they can terminate. For example, the regular expression `.*;` matches any number of characters up to the *last* semicolon it finds. To only match up to the first semicolon, the reluctant version `.*?;` must be used. All greedy operators and the reluctant versions are listed in the following two tables, respectively.

Greedy Operator	Description
`X?`	Matches X zero or one time
`X*`	Matches X zero or more times
`X+`	Matches X one or more times
`X{n}`	Matches X exactly n times, where n is any number
`X{n,}`	Matches X at least n times
`X{n,m}`	Matches X at least n, but no more than m times

Reluctant (Lazy) Operator	Description
`X??`	Matches X zero or one time
`X*?`	Matches X zero or more times
`X+?`	Matches X one or more times
`X{n}?`	Matches X exactly n times, where n is any number
`X{n,}?`	Matches X at least n times
`X{n,m}?`	Matches X at least n, but no more than m times

The language also supports capturing groups of matching characters by using parentheses inside the regular expression. A back reference can be used to reference one of these matching subgroups. A back reference is denoted by a backslash followed by a number corresponding to the number of a subgroup. In the string `(A(B))`, the zero group is the entire expression, then subgroups start numbering after each left parenthesis. Therefore, `A(B)` is the first subgroup, and `B` is the second subgroup. The back references then allow a string to be matched. For example, if you want to match the same word appearing twice in a row, you might use `[([a-zA-Z])\b\1]`. Remember that the `\b` stands for a word boundary. Because the character class for letters is inside parentheses, the text that matched can then be referenced using the back reference meta-character `\1`.

The Pattern Class

The `Pattern` class is responsible for compiling and storing a specified regular expression. There are flags that control how the regular expression is treated. The `regex` is compiled to provide for efficiency. The textual representation of a regular expression is meant for ease of use and understanding by programmers.

Method	Description

`static Pattern compile(String regex)`
`static Pattern compile(String regex, int flags)`
	The compile method accepts a regular expression in a string and compiles it for internal use. The variant form allows you to specify flags that modify how the regular expression is treated.

`static boolean matches(String regex, CharSequence input)`	Compiles a specified regular expression and matches it against the `input`. Returns true if the regular expression describes the input data, and false otherwise. Use this only for quick matches. To match a regular expression repeatedly against different input, the regular expression should be compiled only once.
`static String quote(String s)`	Returns a literal regular expression that will match the string passed in. The returned string starts with `\Q` followed by the string passed in, and ends with `\E`. These are used to quote a string, so what would be meta-characters in the regular expression language are treated literally.

Table continued on following page

Method	Description
`int flags()`	Returns an integer containing the flags set when the regular expression was compiled.
`Matcher matcher(CharSequence input)`	Returns a `Matcher` to use for matching the pattern against the specified input.
`String pattern()`	Returns the regular expression that was used to create the pattern.

`String[] split(CharSequence input)`
`String[] split(CharSequence input, int limit)`
	Returns an array of strings after splitting the input into chunks using the regular expression as a separator. The `limit` can be used to limit how many times the regular expression is matched. The matching text does not get placed into the array. If `limit` is positive, the pattern will be applied at least "limit minus 1" times. If `limit` is 0, the pattern will be applied as many times as it can, and trailing empty strings are removed. If `limit` is negative, the pattern will be applied as many times as it can, and trailing empty strings will be left in the array.

The Matcher Class

The `Matcher` class is used to use a pattern to compare to an input string, and perform a wide variety of useful tasks. The `Matcher` class provides the ability to get a variety of information such as where in the string a pattern matched, replace a matching subset of the string with another string, and other useful operations.

Method	Description
`static String quoteReplacement(String s)`	Returns a string that is quoted with `\Q` and `\E` and can be used to match literally with other input.
`Matcher appendReplacement (StringBuffer sb, String replacement)`	First appends all characters up to a match to the string buffer, then replaces the matching text with `replacement`, then sets the index to one position after the text matched to prepare for the next call to this method. Use `appendTail` to append the rest of the input after the last match.
`StringBuffer appendTail (StringBuffer sb)`	Appends the rest of the input sequence to the string buffer that is passed in.
`MatchResult asResult()`	Returns a reference to a `MatchResult` describing the matcher's state.
`int end()`	Returns the index that is one past the ending position of the last match.

Method	Description
`int end(int group)`	Returns the index that is one past the ending position of a specified capturing group.
`boolean find()`	Returns true if a match is found starting at one index immediately after the previous match, or at the beginning of the line if the matcher has been reset.
`boolean find(int start)`	Resets the matcher and attempts to match the pattern against the input text starting at position `start`. Returns true if a match is found.
`boolean hitEnd()`	Returns true if the end of input was reached by the last match.
`boolean requireEnd()`	Returns true if more input could turn a positive match into a negative match.
`boolean lookingAt()`	Returns true if the pattern matches, but does not require that the pattern has to match the input text completely.
`boolean matches()`	Returns true if the pattern matches the string. The pattern must describe the entire string for this method to return true. For partial matching, use `find()` or `lookingAt()`.
`Pattern pattern()`	Returns a reference to the pattern currently being used on the matcher.
`Matcher reset()`	Resets the matcher's state completely.
`Matcher reset (CharSequence input)`	Resets the matcher's state completely and sets new input to `input`.
`int start()`	Returns the starting position of the previous match.
`int start(int group)`	Returns the starting position of a specified capturing group.
`Matcher usePattern (Pattern newPattern)`	Sets a new pattern to use for matching. The current position in the input is not changed.
`String group()`	Returns a string containing the contents of the previous match.
`String group(int group)`	Returns a string containing the contents of a specific matched group. The 0-th group is always the entire expression.
`int groupCount()`	Returns the number of capturing groups in the matcher's pattern.
`Matcher region(int start, int end)`	Returns a `Matcher` that is confined to a substring of the string to search. The caret and dollar sign meta-characters will match at the beginning and end of the defined region.
`int regionEnd()`	Returns the end index (one past the last position actually checked) of the currently defined region.
`int regionStart()`	Returns the start index of the currently defined region.

Table continued on following page

Method	Description
`String replaceAll(String replacement)`	Replaces all occurrences of the string that match the pattern with the string `replacement`. The `Matcher` should be reset if it will still be used after this method is called.
`String replaceFirst(String replacement)`	Replaces only the first string that matches the pattern with the string `replacement`. The `Matcher` should be reset if it will still be used after this method is called.

The MatchResult Interface

The `MatchResult` interface contains the group methods, and `start` and `end` methods, to provide a complete set of methods allowing for describing the current state of the `Matcher`. The `Matcher` class implements this interface and defines all these methods. The `toMatchResult` method returns a handle to a `MatchResult`, which provides for saving and handling the current state of the `Matcher` class.

Regular Expression Example

Use the `Pattern/Matcher` classes to process a Java source code file. All classes that aren't public will be listed (all classes that have no modifiers, actually), and also all doubled words (such as two identifiers in a row) are listed utilizing back references.

The input source code file (which does not compile) is shown as follows:

```
import java.util.*;

class EmptyClass {
}

class MyArrayList extends extends ArrayList {
}

public class RETestSource {
    public static void main(String args[]) {
        System.out.println("Sample RE test test source code code");
    }
}
```

The program utilizing regular expressions to process this source code follows:

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

public class RegExpExample {

    public static void main(String args[])
    {
        String fileName = "RETestSource.java";

        String unadornedClassRE = "^\\s*class (\\w+)";
```

```
            String doubleIdentifierRE = "\\b(\\w+)\\s+\\1\\b";

            Pattern classPattern = Pattern.compile(unadornedClassRE);
            Pattern doublePattern = Pattern.compile(doubleIdentifierRE);
            Matcher classMatcher, doubleMatcher;

            int lineNumber=0;

            try {
                BufferedReader br = new BufferedReader(new FileReader(fileName));
                String line;

                while( (line=br.readLine()) != null) {
                    lineNumber++;

                    classMatcher = classPattern.matcher(line);
                    doubleMatcher = doublePattern.matcher(line);

                    if(classMatcher.find()) {
                        System.out.println("The class [" +
                                               classMatcher.group(1) +
                                           "] is not public");
                    }

                    while(doubleMatcher.find()) {
                        System.out.println("The word \"" + doubleMatcher.group(1) +
                                           "\" occurs twice at position " +
                                           doubleMatcher.start() + " on line " +
                                           lineNumber);
                    }
                }
            } catch(IOException ioe) {
                System.out.println("IOException: " + ioe);
                ioe.printStackTrace();
            }
        }
    }
```

The first regular expression, `^\\s*class (\\w+)`, searches for unadorned `class` keywords starting at the beginning of the line, followed by zero or more white space characters, then the literal `class`. The group operator is used with one or more word characters (A–Z, a–z, 0–9, and the underscore), so the class name gets matched.

The second regular expression, `\\b(\\w+)\\s+\\1\\b`, uses the word boundary meta-character (`\b`) to ensure that words are isolated. Without this, the string `public class` would match on the letter `c`. A back reference is used to match a string already matched, in this case, one or more word characters. One or more characters of white space must appear between the words. Executing the previous program on the preceding test Java source file gives you the following output:

```
The class [EmptyClass] is not public
The class [MyArrayList] is not public
The word "extends" occurs twice at position 18 on line 6
The word "test" occurs twice at position 32 on line 11
The word "code" occurs twice at position 49 on line 11
```

Java Preferences

Programs commonly must store configuration information in some manner that is easy to change and external to the program itself. Java offers utility classes for storing and retrieving system-defined and user-defined configuration information. There are separate hierarchies for the user and system information. All users share the preference information defined in the system tree; each user has his or her own tree for configuration data isolated from other users. This allows for custom configuration, including overriding system values.

The core of the preferences class library is the abstract class `java.util.prefs.Preferences`. This class defines a set of methods that provides for all the features of the preferences library.

Each node in a preference hierarchy has a name, which does not have to be unique. The root node of a preference tree has the empty string ("") as its name. The forward slash is used as a separator for the names of preference nodes, much like it is used as a separator for directory names on Unix. The only two strings that are not valid node names are the empty string (because it is reserved for the root node) and a forward slash by itself (because it is a node separator). The root node's path is the forward slash by itself. Much like with directories, absolute and relative paths are possible. An absolute path always starts with a forward slash, because the absolute path always starts at the root node and follows the tree down to a specific node. A relative path never starts with a forward slash. A path is valid as long as there aren't two consecutive forward slashes in the pathname, and no path except the path to root ends in the forward slash.

Because preferences are implemented by a third-party implementer, changes to the preferences aren't always immediately written to the backing store.

The maximum length of a single node's name and any of its keys is 80 characters. The maximum length of a string value in a node is 8,192 characters.

The Preferences Class

The `Preferences` class is the main class used for dealing with preferences. It represents a node in the preference's tree and contains a large number of methods to manipulate this tree and also nodes in the tree. It is basically a one-stop shop for using preferences. The following sections outline the `Preferences` methods.

Operations on the Preferences Tree

The `Preferences` class defines a number of methods that allow for the creation/deletion of nodes and the retrieval of certain nodes in the tree.

Method	Description
`Preferences node(String pathName)`	Returns a specified node. If the node does not exist, it is created (and any ancestors that do not exist are created) and returned.
`boolean nodeExists(String pathName)`	Returns true if the path to a node exists in the current tree. The path can be an absolute or relative path.

Method	Description
`void removeNode()`	Removes this preference node and all of its children. The only methods that can be invoked after a node has been removed are `name()`, `absolutePath()`, `isUserNode()`, `flush()`, and `nodeExists("")`, and those inherited from `Object`. All other methods will throw an `IllegalStateException`. The removal may not be permanent until `flush()` is called to persist the changes to the tree.
`static Preferences systemNodeForPackage(Class c)`	This method returns a preference node for the package that the specified class is in. All periods in the package name are replaced with forward slashes. For a class that has no package, the name of the node that is returned is literally `<unnamed>`. This node should not be used long term, because it is shared by all programs that use it. If the node does not already exist, the node and all ancestors that do not exist will automatically be created.
`static Preferences systemRoot()`	This method returns the root node for the system preferences tree.
`static Preferences userNodeForPackage(Class c)`	This method returns a preference node for the package that the specified class is in. All periods in the package name are replaced with forward slashes. For a class that has no package, the name of the node that is returned is literally `<unnamed>`. This node should not be used long term, because it is shared by all programs that use it, so configuration settings are not isolated. If the node does not already exist, the node and all ancestors that do not exist will automatically get created.
`static Preferences userRoot()`	This method returns the root node for the user preferences tree.

Retrieving Information about the Node

Each node has information associated with it, such as its path, parent and children nodes, and the node's name. The methods to manipulate this information are shown here.

Method	Description
`String absolutePath()`	This method returns the absolute path to the current node. The absolute path starts at the root node, `/`, and continues to the current node.
`String[] childrenNames()`	Returns an array of the names of all child nodes of the current node.
`boolean isUserNode()`	Returns true if this node is part of the user configuration tree, or false if this node is part of the system configuration tree.
`String name()`	Returns the name of the current node.
`Preferences parent()`	Returns a `Preferences` reference to the parent of the current node, or null if trying to get the parent of the root node.

Retrieving Preference Values from the Node

The following methods act much like those from the `Hashtable` class. The key difference is that there are versions of the `get` for most primitive types. Each type is associated with a specific key, a string standing for the name of the configuration parameter.

Method	Description
`String[] keys()`	Returns an array of strings that contains the names of all keys in the current preferences node.
`String get(String key, String def)`	Returns the string associated with a specified key. If the key does not exist, it is created with the default value `def` and this default value is then returned.
`boolean getBoolean (String key, boolean def)`	Returns the `boolean` associated with a specified key. If the key does not exist, it is created with the default value `def` and this default value is then returned.
`byte[] getByteArray (String key, byte[] def)`	Returns the `byte` array associated with a specified key. If the key does not exist, it is created with the default value `def` and this default value is then returned.
`double getDouble(String key, double def)`	Returns the `double` associated with a specified key. If the key does not exist, it is created with the default value `def` and this default value is then returned.
`float getFloat(String key, float def)`	Returns the `float` associated with a specified key. If the key does not exist, it is created with the default value `def` and this default value is then returned.

Method	Description
`int getInt(String key, int def)`	Returns the `integer` associated with a specified key. If the key does not exist, it is created with the default value `def` and this default value is then returned.
`long getLong(String key, long def)`	Returns the `long` associated with a specified key. If the key does not exist, it is created with the default value `def` and this default value is then returned.

Setting Preference Values on the Node

Along with each `get` method is a `put` version intended for setting the information associated with a given configuration parameter's key name.

Method	Description

`void put(String key, String value)`
`void putBoolean(String key, boolean value)`
`void putByteArray(String key, byte[] value)`
`void putDouble(String key, double value)`
`void putInt(String key, int value)`
`void putFloat(String key, float value)`
`void putLong(String key, long value)`
	These methods set a configuration parameter (the name of which is passed in as `key`) to a specific type. If `key` or `value` is null, an exception is thrown. The key can be at most 80 characters long (defined in `MAX_KEY_LENGTH`) and the value can be at most 8,192 characters (defined in `MAX_VALUE_LENGTH`).

Events

Two events are defined for the `Preference` class — one fires when a node is changed in the preference tree, and the second fires when a preference is changed. The methods for these events are listed in the next table.

Method	Description
`void addNodeChangeListener (NodeChangeListener ncl)`	Adds a listener for notification of when a child node is added or removed from the current preference node.
`void addPreferenceChangeListener (PreferenceChangeListener pcl)`	Adds a listener for preference change events — anytime a preference is added to, removed from, or the value is changed, listeners will be notified.
`void removeNodeChangeListener (NodeChangeListener ncl)`	Removes a specified node change listener.
`void removePreferenceChangeListener (PreferenceChangeListener pcl)`	Removes a specified preference change listener.

Other Operations

The following table lists the other methods in the `Preference` class, such as writing any pending changes to the backing store, resetting the preference hierarchy to empty, saving the hierarchy to disk, and other operations.

Method	Description
`void clear()`	Removes all preferences on this node.
`void exportNode (OutputStream os)`	Writes the entire contents of the node (and only the current node) to the output stream as an XML file (following the `preferences .dtd` listed in the following section).
`void exportSubtree (OutputStream os)`	Writes the entire contents of this node and all nodes located below this node in the preferences tree to the output stream as an XML file (following the `preferences.dtd` listed in the following section).
`void flush()`	Writes any changes to the preference node to the backing store, including data on all children nodes.
`void remove(String key)`	Removes the value associated with the specified key.
`void sync()`	Ensures that the current version of the preference node in memory matches that of the stored version. If data in the preference node needs to be written to the backing store, it will be.
`String toString()`	Returns a string containing `User` or `System`, depending on which hierarchy the node is in, and the absolute path to the current node.

Exporting to XML

The `Preferences` system defines a standard operation to export the entire tree of keys/values to an XML file. This XML file's DTD is available at `http://java.sun.com/dtd/preferences.dtd`. This DTD is also included here:

```
<?xml version="1.0" encoding="UTF-8"?>

    <!-- DTD for a Preferences tree. -->

    <!-- The preferences element is at the root of an XML document
         representing a Preferences tree. -->
    <!ELEMENT preferences (root)>

    <!-- The preferences element contains an optional version
         attribute, which specifies version of DTD. -->
    <!ATTLIST preferences EXTERNAL_XML_VERSION CDATA "0.0" >

    <!-- The root element has a map representing the root's preferences
        (if any), and one node for each child of the root (if any). -->
    <!ELEMENT root (map, node*) >

    <!-- Additionally, the root contains a type attribute, which
         specifies whether it's the system or user root. -->
```

```
      <!ATTLIST root
                type (system|user) #REQUIRED >

      <!-- Each node has a map representing its preferences (if any),
           and one node for each child (if any). -->
      <!ELEMENT node (map, node*) >

      <!-- Additionally, each node has a name attribute -->
      <!ATTLIST node
                name CDATA #REQUIRED >

     <!-- A map represents the preferences stored at a node (if any). -->
      <!ELEMENT map (entry*) >

      <!-- An entry represents a single preference, which is simply
            a key-value pair. -->
      <!ELEMENT entry EMPTY >
      <!ATTLIST entry
                key   CDATA #REQUIRED
                value CDATA #REQUIRED >
```

Using Preferences

The following example sets a few properties in a node in the user tree, prints out information about the node, and then exports the information to an XML file:

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

   public class PreferenceExample {
       public void printInformation(Preferences p)
          throws BackingStoreException
       {
           System.out.println("Node's absolute path: " + p.absolutePath());

           System.out.print("Node's children: ");
           for(String s : p.childrenNames()) {
               System.out.print(s + " ");
           }
           System.out.println("");

           System.out.print("Node's keys: ");
           for(String s : p.keys()) {
               System.out.print(s + " ");
           }
           System.out.println("");

           System.out.println("Node's name: " + p.name());
           System.out.println("Node's parent: " + p.parent());
           System.out.println("NODE: " + p);
           System.out.println("userNodeForPackage: " +
                   Preferences.userNodeForPackage(PreferenceExample.class));
           System.out.println("All information in node");
```

```
            for(String s : p.keys()) {
                System.out.println("  " + s + " = " + p.get(s, ""));
            }
        }

        public void setSomeProperties(Preferences p)
           throws BackingStoreException
        {
            p.put("fruit", "apple");
            p.put("cost", "1.01");
            p.put("store", "safeway");
        }

        public void exportToFile(Preferences p, String fileName)
           throws BackingStoreException
        {
            try {
                FileOutputStream fos = new FileOutputStream(fileName);

                p.exportSubtree(fos);
                fos.close();
            } catch(IOException ioe) {
                System.out.println("IOException in exportToFile\n" + ioe);
                ioe.printStackTrace();
            }
        }

        public static void main(String args[])
        {
            PreferenceExample pe = new PreferenceExample();
            Preferences prefsRoot = Preferences.userRoot();
            Preferences myPrefs = prefsRoot.node("PreferenceExample");

            try {
                pe.setSomeProperties(myPrefs);
                pe.printInformation(myPrefs);
                pe.exportToFile(myPrefs, "prefs.xml");
            } catch(BackingStoreException bse) {
                System.out.println("Problem with accessing the backing store\n" + bse);
                bse.printStackTrace();
            }
        }
    }
```

The output to the screen is shown here:

```
Node's absolute path: /PreferenceExample
Node's children:
Node's keys: fruit cost store
Node's name: PreferenceExample
Node's parent: User Preference Node: /
NODE: User Preference Node: /PreferenceExample
```

```
userNodeForPackage: User Preference Node: /<unnamed>
All information in node
  fruit = apple
  cost = 1.01
  store = safeway
```

The exported information in the XML file is listed here:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE preferences SYSTEM "http://java.sun.com/dtd/preferences.dtd">
<preferences EXTERNAL_XML_VERSION="1.0">
  <root type="user">
    <map/>
    <node name="PreferenceExample">
      <map>
        <entry key="fruit" value="apple"/>
        <entry key="cost" value="1.01"/>
        <entry key="store" value="safeway"/>
      </map>
    </node>
  </root>
</preferences>
```

Summary

This chapter introduced Derby, a lightweight database from the Apache Database project, which is new in JDK 6. Also reviewed were the new language features that Sun built into the JDK 5 release of the Java programming language. You should have all you need to know to understand and utilize these new features. You may find that a number of programming tasks you've accomplished in the past are now made simpler and clearer, and perhaps even some problems that never had a good solution now do.

Also covered in this chapter are several of the most important utility libraries in Java. The preferences library allows you to store and retrieve configuration information for your application. The logging library provides a sophisticated package of routines to track what your program is doing and offer output in a variety of ways. The regular expression library provides routines for advanced processing of textual data.

Now that you have learned about the advanced language features in Java, the next two chapters take you inside a modern Java development shop. In Chapter 2, the habits, tools, and methodologies that make an effective Java developer are discussed.