Chapter 7

Exploring the Basic APIs Part 2

Chapter 7 continues to explore Java's basic (language-oriented) APIs by introducing APIs that let you use reflection to obtain type information at runtime and more, manage strings, perform system activities (such as retrieving a system property value and obtaining the current time), and use threads to improve application performance.

NOTE: Chapter 7 explores basic API classes and interfaces that are located in the `java.lang` and `java.lang.reflect` packages.

Reflection API

Chapter 3 referred to *reflection* as a third form of runtime type identification (RTTI). Java's Reflection API lets applications learn about loaded classes, interfaces, enums (a kind of class), and annotation types (a kind of interface). The API also lets applications instantiate classes, call methods, access fields, and perform other tasks reflectively.

Chapter 5 presented a `StubFinder` application that used part of the Reflection API to load a class and identify all of the loaded class's public methods that are annotated with `@Stub` annotations. This tool is one example where using reflection is beneficial. Another example is the *class browser*, a tool that enumerates the members of a class.

CAUTION: Reflection should not be used indiscriminately. Application performance suffers because it takes longer to perform operations with reflection than without reflection. Also, reflection-oriented code can be harder to read, and the absence of compile-time type checking can result in runtime failures.

The `java.lang` package's `Class` class is the entry point into the Reflection API. `Class` is generically declared as `Class<T>`, where `T` identifies the class, interface, enum, or annotation type that is being modeled by the `Class` object. `T` can be replaced by `?` (as in `Class<?>`) when the type being modeled is unknown.

Table 7–1 describes some of `Class`'s methods.

Table 7–1. *`Class` Methods*

Method	Description
`static Class<?> forName(String typename)`	Return the `Class` object that is associated with `typename`, which must include the type's qualified package name when the type is part of a package (`java.lang.String`, for example). If the class or interface type has not been loaded into memory, this method takes care of *loading* (reading the classfile's contents into memory), *linking* (taking these contents and combining them into the runtime state of the virtual machine so that they can be executed), and *initializing* (setting class fields to default values, running class initializers, and performing other class initialization) prior to returning the `Class` object. This method throws `java.lang.ClassNotFoundException` when the type cannot be found, `java.lang.LinkageError` when an error occurs during linkage, and `java.lang.ExceptionInInitializerError` when an exception occurs during a class's static initialization.
`Annotation[] getAnnotations()`	Return a possibly empty array containing all annotations that are declared for the class represented by this `Class` object.
`Constructor[] getConstructors()`	Return an array containing `Constructor` objects representing all the public constructors of the class represented by this `Class` object. An array of length zero is returned when the represented class has no public constructors, this `Class` object represents an array class, or this `Class` object represents a primitive type or void.
`Annotation[] getDeclaredAnnotations()`	Return an array containing all annotations that are directly declared on the class represented by this `Class` object—inherited annotations are not included. The returned array might be empty.
`Constructor[] getDeclaredConstructors()`	Return an array of `Constructor` objects representing all the constructors declared by the class represented by this `Class` object. These are public, protected, default (package) access, and private constructors. The elements in the returned array are not sorted and are not in any particular order. If the class has a default constructor, it is included in the returned array. This method returns an array of length zero when this `Class` object represents an interface, a primitive type, an array class, or void.

Method	Description
`Field[] getDeclaredFields()`	Return an array of `Field` objects representing all the fields declared by the class or interface represented by this `Class` object. This array includes public, protected, default (package) access, and private fields, but excludes inherited fields. The elements in the returned array are not sorted and are not in any particular order. This method returns an array of length zero when the class or interface declares no fields, or when this `Class` object represents a primitive type, an array class, or void.
`Method[] getDeclaredMethods()`	Return an array of `Method` objects representing all the methods declared by the class or interface represented by this `Class` object. This array includes public, protected, default (package) access, and private methods, but excludes inherited methods. The elements in the returned array are not sorted and are not in any particular order. This method returns an array of length zero when the class or interface declares no methods, or when this `Class` object represents a primitive type, an array class, or void.
`Field[] getFields()`	Return an array containing `Field` objects representing all the public fields of the class or interface represented by this `Class` object, including those public fields inherited from superclasses and superinterfaces. The elements in the returned array are not sorted and are not in any particular order. This method returns an array of length zero when this `Class` object represents a class or interface that has no accessible public fields, or when this `Class` object represents an array class, a primitive type, or void.
`Method[] getMethods()`	Return an array containing `Method` objects representing all the public methods of the class or interface represented by this `Class` object, including those public methods inherited from superclasses and superinterfaces. Array classes return all the public member methods inherited from the `Object` class. The elements in the returned array are not sorted and are not in any particular order. This method returns an array of length zero when this `Class` object represents a class or interface that has no public methods, or when this `Class` object represents a primitive type or void.
`String getName()`	Return the name of the class represented by this `Class` object.
`Package getPackage()`	Return a `Package` object that describes the package in which the class represented by this `Class` object is located, or null when the class is a member of the unnamed package.

Method	Description
`Class<? super T> getSuperclass()`	Return the `Class` object representing the superclass of the entity (class, interface, primitive type, or void) represented by this `Class` object. When the `Class` object on which this method is called represents the `Object` class, an interface, a primitive type, or void, null is returned. When this object represents an array class, the `Class` object representing the `Object` class is returned.
`T newInstance()`	Create and return a new instance of the class represented by this `Class` object. The class is instantiated as if by a `new` expression with an empty argument list. The class is initialized when it has not already been initialized. This method throws `java.lang.IllegalAccessException` when the class or its noargument constructor is not accessible; `java.lang.InstantiationException` when this `Class` object represents an abstract class, an interface, an array class, a primitive type, or void, or when the class does not have a noargument constructor (or when instantiation fails for some other reason); and `ExceptionInInitializerError` when initialization fails because the object threw an exception during initialization.

Table 7–1's description of the `forName()` method reveals one way to obtain a `Class` object. This method loads, links, and initializes a class or interface that is not in memory and returns a `Class` object that represents the class or interface. Listing 7–1 demonstrates `forName()` and additional methods described in this table.

Listing 7–1. ***Using reflection to identify a class's name, package, public fields, constructors, and methods***

```
import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.Method;

public class ExploreType
{
   public static void main(String[] args)
   {
      if (args.length != 1)
      {
         System.err.println("usage: java ExploreType pkgAndTypeName");
         return;
      }
      try
      {
         Class<?> clazz = Class.forName(args[0]);
         System.out.println("NAME: " + clazz.getName());
         System.out.println("PACKAGE: " + clazz.getPackage().getName());
         System.out.println("FIELDS");
         Field[] fields = clazz.getDeclaredFields();
         for (int i = 0; i < fields.length; i++)
            System.out.println(fields[i]);
         System.out.println("CONSTRUCTORS");
         Constructor[] constructors = clazz.getDeclaredConstructors();
         for (int i = 0; i < constructors.length; i++)
            System.out.println(constructors[i]);
         System.out.println("METHODS");
         Method[] methods = clazz.getDeclaredMethods();
         for (int i = 0; i < methods.length; i++)
            System.out.println(methods[i]);
      }
      catch (ClassNotFoundException cnfe)
      {
         System.err.println("could not locate " + args[0]);
      }
   }
}
```

Listing 7–1 presents an application that uses the Reflection API to explore a class or interface by outputting its name, package, fields, constructors (classes only), and methods. Only fields, constructors, and methods that are declared in the class, or fields and methods that are declared in the interface, are output.

After verifying that one command-line argument has been passed to this application, `main()` calls `forName()` to try to return a `Class` object representing the class or interface identified by this argument. If successful, the returned object's reference is assigned to `clazz`—I cannot name this variable `class` because `class` is a reserved word.

`forName()` throws an instance of the checked `ClassNotFoundException` class when it cannot locate the class's classfile (perhaps the classfile was erased prior to executing the application). It also throws `LinkageError` when a class's classfile is malformed, and `ExceptionInInitializerError` when a class's static initialization fails.

NOTE: `ExceptionInInitializerError` is often thrown as the result of a class initializer throwing an unchecked exception. For example, the class initializer in the following `FailedInitialization` class results in `ExceptionInInitializerError` because `someMethod()` throws `NullPointerException`:

```
public class FailedInitialization
{
   static
   {
      someMethod(null);
   }
   public static void someMethod(String s)
   {
      int len = s.length(); // s contains null
      System.out.println(s + "'s length is " + len + " characters");
   }
}
```

When you run this application, you must include the package specification when the class or interface is located in a package. For example, specifying `java ExploreType java.lang.Boolean` to output the fields, constructors, and methods declared in the `java.lang` package's `Boolean` class results in the following output:

```
NAME: java.lang.Boolean
PACKAGE: java.lang
FIELDS
public static final java.lang.Boolean java.lang.Boolean.TRUE
public static final java.lang.Boolean java.lang.Boolean.FALSE
public static final java.lang.Class java.lang.Boolean.TYPE
private final boolean java.lang.Boolean.value
private static final long java.lang.Boolean.serialVersionUID
CONSTRUCTORS
public java.lang.Boolean(java.lang.String)
public java.lang.Boolean(boolean)
METHODS
public int java.lang.Boolean.hashCode()
public boolean java.lang.Boolean.equals(java.lang.Object)
public int java.lang.Boolean.compareTo(java.lang.Boolean)
public int java.lang.Boolean.compareTo(java.lang.Object)
public static boolean java.lang.Boolean.getBoolean(java.lang.String)
public static java.lang.String java.lang.Boolean.toString(boolean)
public java.lang.String java.lang.Boolean.toString()
public static java.lang.Boolean java.lang.Boolean.valueOf(java.lang.String)
public static java.lang.Boolean java.lang.Boolean.valueOf(boolean)
public boolean java.lang.Boolean.booleanValue()
public static boolean java.lang.Boolean.parseBoolean(java.lang.String)
private static boolean java.lang.Boolean.toBoolean(java.lang.String)
```

Table 7–1's descriptions of the `getAnnotations()` and `getDeclaredAnnotations()` methods reveal that each method returns an array of `Annotation`, an interface that is located in the `java.lang.annotation` package. `Annotation` is the superinterface of `Override`, `SuppressWarnings`, and all other annotation types.

Table 7–1's method descriptions also refer to `Constructor`, `Field`, and `Method`. Instances of these classes (which are members of the `java.lang.reflect` package) represent a class's constructors and a class's or an interface's fields and methods.

`Constructor` represents a constructor and is generically declared as `Constructor<T>`, where `T` identifies the class in which the constructor represented by `Constructor` is declared. `Constructor` declares various methods, including the following methods:

- `Annotation[] getDeclaredAnnotations()` returns an array of all annotations declared on the constructor. The returned array has zero length when there are no annotations.
- `Class<T> getDeclaringClass()` returns a `Class` object that represents the class in which the constructor is declared.

- `Class[]<?> getExceptionTypes()` returns an array of `Class` objects representing the types of exceptions listed in the constructor's throws clause. The returned array has zero length when there is no throws clause.
- `String getName()` returns the constructor's name.
- `Class[]<?> getParameterTypes()` returns an array of `Class` objects representing the constructor's parameters. The returned array has zero length when the constructor does not declare parameters.

`Field` represents a field and declares various methods, including the following methods:

- `Object get(Object object)` returns the value of the field for the specified `object`.
- `boolean getBoolean(Object object)` returns the value of the Boolean field for the specified `object`.
- `byte getByte(Object object)` returns the value of the byte integer field for the specified `object`.
- `char getChar(Object object)` returns the value of the character field for the specified `object`.
- `double getDouble(Object object)` returns the value of the double precision floating-point field for the specified `object`.
- `float getFloat(Object object)` returns the value of the floating-point field for the specified `object`.
- `int getInt(Object object)` returns the value of the integer field for the specified `object`.
- `long getLong(Object object)` returns the value of the long integer field for the specified `object`.
- `short getShort(Object object)` returns the value of the short integer field for the specified `object`.

`get()` returns the value of any type of field. In contrast, the other listed methods return the values of specific types of fields. All of these methods throw `NullPointerException` when `object` is `null` and the field is an instance field, `IllegalArgumentException` when `object` is not an instance of the class or interface declaring the underlying field (or not an instance of a subclass or interface implementor), and `IllegalAccessException` when the underlying field cannot be accessed (it is private, for example).

`Method` represents a method and declares various methods, including the following methods:

- `int getModifiers()` returns a 32-bit integer whose bit fields identify the method's reserved word modifiers (such as `public`, `abstract`, or `static`). These bit fields must be interpreted via the `java.lang.reflect.Modifier` class. For example, you might specify (`method.getModifiers() & Modifier.ABSTRACT) == Method.ABSTRACT` to find out if the method (represented by the `Method` object whose reference is stored in `method`) is abstract—this expression evaluates to true when the method is abstract.
- `Class<?> getReturnType()` returns a `Class` object that represents the method's return type.
- `Object invoke(Object receiver, Object... args)` calls the method on the object identified by `receiver` (which is ignored when the method is a class method), passing the variable number of arguments identified by `args` to the called method. The `invoke()` method throws `NullPointerException` when `receiver` is `null` and the method being called is an instance method, `IllegalAccessException` when the method is not accessible (it is private, for example), `IllegalArgumentException` when an incorrect number of arguments are passed to the method (and other reasons), and `java.lang.reflect.InvocationTargetException` when an exception is thrown from the called method.
- `boolean isVarArgs()` returns true when the method is declared to receive a variable number of arguments.

The `java.lang.reflect.AccessibleObject` class is the superclass of `Constructor`, `Field`, and `Method`. This superclass provides methods for reporting a constructor's, field's, or method's accessibility (is it private?) and making an inaccessible constructor, field, or method accessible. `AccessibleObject`'s methods include the following:

- `T getAnnotation(Class<T> annotationType)` returns the constructor's, field's, or method's annotation of the specified type when such an annotation is present; otherwise, null returns.
- `boolean isAccessible()` returns true when the constructor, field, or method is accessible.
- `boolean isAnnotationPresent(Class<? extends Annotation> annotationType)` returns true when an annotation of the type specified by `annotationType` has been declared on the constructor, field, or method. This method takes inherited annotations into account.
- `void setAccessible(boolean flag)` attempts to make an inaccessible constructor, field, or method accessible when `flag` is `true`.

NOTE: The `java.lang.reflect` package also includes an `Array` class whose class methods make it possible to reflectively create and access Java arrays.

Another way to obtain a `Class` object is to call `Object`'s `getClass()` method on an object reference; for example, `Employee e = new Employee(); Class<? extends Employee> clazz = e.getClass();`. The `getClass()` method does not throw an exception because the class from which the object was created is already present in memory.

There is one more way to obtain a `Class` object, and that is to employ a *class literal*, which is an expression consisting of a class name, followed by a period separator, followed by reserved word `class`. Examples of class literals include `Class<Employee> clazz = Employee.class;` and `Class<String> clazz = String.class`.

Perhaps you are wondering about how to choose between `forName()`, `getClass()`, and a class literal. To help you make your choice, the following list compares each competitor:

- `forName()` is very flexible in that you can dynamically specify any reference type by its package-qualified name. If the type is not in memory, it is loaded, linked, and initialized. However, lack of compile-time type safety can lead to runtime failures.
- `getClass()` returns a `Class` object describing the type of its referenced object. If called on a superclass variable containing a subclass instance, a `Class` object representing the subclass type is returned. Because the class is in memory, type safety is assured.
- A class literal returns a `Class` object representing its specified class. Class literals are compact and the compiler enforces type safety by refusing to compile the source code when it cannot locate the literal's specified class.

NOTE: You can use class literals with primitive types, including void. Examples include `int.class`, `double.class`, and `void.class`. The returned `Class` object represents the class identified by a primitive wrapper class's `TYPE` field or `java.lang.Void.TYPE`. For example, each of `int.class == Integer.TYPE` and `void.class == Void.TYPE` evaluates to true.

You can also use class literals with primitive type–based arrays. Examples include `int[].class` and `double[].class`. For these examples, the returned `Class` objects represent `Class<int[]>` and `Class<double[]>`.

String Management

Many computer languages implement the concept of a *string*, a sequence of characters treated as a single unit (and not as individual characters). For example, the C language implements a string as an array of characters terminated by the null character (`'\0'`). In contrast, Java implements a string via the `java.lang.String` class.

String objects are immutable: you cannot modify a String object's string. The various String methods that appear to modify the String object actually return a new String object with modified string content instead. Because returning new String objects is often wasteful, Java provides the java.lang.StringBuffer class as a workaround.

This section introduces you to String and StringBuffer.

String

String represents a string as a sequence of characters. Unlike C strings, this sequence is not terminated by a null character. Instead, its length is stored separately.

The Java language provides syntactic sugar that simplifies working with strings. For example, the compiler recognizes `String favLanguage = "Java";` as the assignment of string literal "Java" to String variable favLanguage. Without this sugar, you would have to specify `String favLanguage = new String("Java");`.

Table 7–2 describes some of String's constructors and methods for initializing String objects and working with strings.

Table 7–2. *String Constructors and Methods*

Method	Description
`String(char[] data)`	Initialize this String object to the characters in the data array. Modifying data after initializing this String object has no effect on the object.
`String(String s)`	Initialize this String object to s's string.
`char charAt(int index)`	Return the character located at the zero-based index in this String object's string. This method throws java.lang.IndexOutOfBoundsException when index is less than 0 or greater than or equal to the length of the string.
`String concat(String s)`	Return a new String object containing this String object's string followed by the s argument's string.
`boolean endsWith(String suffix)`	Return true when this String object's string ends with the characters in the suffix argument, when suffix is empty (contains no characters), or when suffix contains the same character sequence as this String object's string. This method performs a case-sensitive comparison (a is not equal to A, for example), and throws NullPointerException when suffix is null.
`boolean equals(Object object)`	Return true when object is of type String and this argument's string contains the same characters (and in the same order) as this String object's string.

Method	Description
`boolean equalsIgnoreCase(String s)`	Return true when `s` and this `String` object contain the same characters (ignoring case). This method returns false when the character sequences differ or when `null` is passed to `s`.
`int indexOf(int c)`	Return the zero-based index of the first occurrence (from the start of the string to the end of the string) of the character represented by `c` in this `String` object's string. Return -1 when this character is not present.
`int indexOf(String s)`	Return the zero-based index of the first occurrence (from the start of the string to the end of the string) of `s`'s character sequence in this `String` object's string. Return -1 when `s` is not present. This method throws `NullPointerException` when `s` is `null`.
`String intern()`	Search an internal table of `String` objects for an object whose string is equal to this `String` object's string. This `String` object's string is added to the table when not present. Return the object contained in the table whose string is equal to this `String` object's string. The same `String` object is always returned for strings that are equal.
`int lastIndexOf(int c)`	Return the zero-based index of the last occurrence (from the start of the string to the end of the string) of the character represented by `c` in this `String` object's string. Return -1 when this character is not present.
`int lastIndexOf(String s)`	Return the zero-based index of the last occurrence (from the start of the string to the end of the string) of `s`'s character sequence in this `String` object's string. Return -1 when `s` is not present. This method throws `NullPointerException` when `s` is `null`.
`int length()`	Return the number of characters in this `String` object's string.
`String replace(char oldChar, char newChar)`	Return a new `String` object whose string matches this `String` object's string except that all occurrences of `oldChar` have been replaced by `newChar`.
`String[] split(String expr)`	Split this `String` object's string into an array of `String` objects using the *regular expression* (a string whose *pattern* [template] is used to search a string for substrings that match the pattern) specified by `expr` as the basis for the split. This method throws `NullPointerException` when `expr` is `null` and `java.util.regex.PatternSyntaxException` when `expr`'s syntax is invalid.

Method	Description
`boolean startsWith(String prefix)`	Return true when this `String` object's string starts with the characters in the `prefix` argument, when `prefix` is empty (contains no characters), or when `prefix` contains the same character sequence as this `String` object's string. This method performs a case-sensitive comparison (a is not equal to A, for example), and throws `NullPointerException` when `prefix` is `null`.
`String substring(int start)`	Return a new `String` object whose string contains this `String` object's characters beginning with the character located at `start`. This method throws `IndexOutOfBoundsException` when `start` is negative or greater than the length of this `String` object's string.
`char[] toCharArray()`	Return a character array that contains the characters in this `String` object's string.
`String toLowerCase()`	Return a new `String` object whose string contains this `String` object's characters where uppercase letters have been converted to lowercase. This `String` object is returned when it contains no uppercase letters to convert.
`String toUpperCase()`	Return a new `String` object whose string contains this `String` object's characters where lowercase letters have been converted to uppercase. This `String` object is returned when it contains no lowercase letters to convert.
`String trim()`	Return a new `String` object that contains this `String` object's string with *whitespace characters* (characters whose Unicode values are 32 or less) removed from the start and end of the string, or this `String` object if no leading/trailing whitespace.

Table 7–2 reveals a couple of interesting items about `String`. First, this class's `public String(String s)` constructor does not initialize a `String` object to a string literal. Instead, it initializes the `String` object to the contents of another `String` object. This behavior suggests that a string literal is more than what it appears to be.

In reality, a string literal is a `String` object. You can prove this to yourself by executing `System.out.println("abc".length());` and `System.out.println("abc" instanceof String);`. The first method call outputs `3`, which is the length of the `"abc" String` object's string, and the second method call outputs `true` (`"abc"` is a `String` object).

The second interesting item is the `intern()` method, which *interns* (stores a unique copy of) a `String` object in an internal table of `String` objects. `intern()` makes it possible to compare strings via their references and `==` or `!=`. These operators are the fastest way to compare strings, which is especially valuable when sorting a huge number of strings.

By default, `String` objects denoted by literal strings (`"abc"`) and string-valued constant expressions (`"a" + "bc"`) are interned in this table, which is why `System.out.println("abc" == "a" + "bc");` outputs `true`. However, `String` objects created via `String` constructors are not interned, which is why `System.out.println("abc" == new String("abc"));` outputs `false`. In contrast, `System.out.println("abc" == new String("abc").intern());` outputs `true`.

Table 7–2 also reveals `split()`, a method that I employed in Chapter 5's `StubFinder` application to split a string's comma-separated list of values into an array of `String` objects. This method uses a regular expression that identifies a sequence of characters around which the string is split. (I will discuss regular expressions in Chapter 9.)

TIP: The `charAt()` and `length()` methods are useful for iterating over a string's characters. For example, `String s = "abc"; for (int i = 0; i < s.length(); i++) System.out.println(s.charAt(i));` returns each of `s`'s `a`, `b`, and `c` characters and outputs each character on a separate line.

StringBuffer

`StringBuffer` provides an internal character array for building a string efficiently. After creating a `StringBuffer` object, you call various methods to append, delete, and insert the character representations of various values to, from, and into the array. You then call `toString()` to convert the array's content to a `String` object and return this object.

CAUTION: Divulging a class's internal implementation typically is not a good idea because doing so violates information hiding. Furthermore, the internal implementation might change, which voids the description of the previous implementation. However, I believe that divulging `StringBuffer`'s internal array adds value to my discussion of this class. Furthermore, it is highly unlikely that `StringBuffer` will ever use anything other than a character array.

Table 7–3 describes some of `StringBuffer`'s constructors and methods for initializing `StringBuffer` objects and working with string buffers.

Table 7–3. *StringBuffer Constructors and Methods*

Method	Description
StringBuffer()	Initialize this StringBuffer object to an empty array with an initial capacity of 16 characters.
StringBuffer(int capacity)	Initialize this StringBuffer object to an empty array with an initial capacity of capacity characters. This constructor throws java.lang.NegativeArraySizeException when capacity is negative.
StringBuffer(String s)	Initialize this StringBuffer object to an array containing s's characters. This object's initial capacity is 16 plus the length of s. This constructor throws NullPointerException when s is null.
StringBuffer append(boolean b)	Append "true" to this StringBuffer object's array when b is true and "false" to the array when b is false, and return this StringBuffer object.
StringBuffer append(char ch)	Append ch's character to this StringBuffer object's array, and return this StringBuffer object.
StringBuffer append(char[] chars)	Append the characters in the chars array to this StringBuffer object's array, and return this StringBuffer object. This method throws NullPointerException when chars is null.
StringBuffer append(double d)	Append the string representation of d's double precision floating-point value to this StringBuffer object's array, and return this StringBuffer object.
StringBuffer append(float f)	Append the string representation of f's floating-point value to this StringBuffer object's array, and return this StringBuffer object.
StringBuffer append(int i)	Append the string representation of i's integer value to this StringBuffer object's array, and return this StringBuffer object.
StringBuffer append(long l)	Append the string representation of l's long integer value to this StringBuffer object's array, and return this StringBuffer object.
StringBuffer append(Object obj)	Call obj's toString() method and append the returned string's characters to this StringBuffer object's array. Append "null" to the array when null is passed to obj. Return this StringBuffer object.
StringBuffer append(String s)	Append s's string to this StringBuffer object's array. Append "null" to the array when null is passed to s. Return this StringBuffer object.

Method	Description
`int capacity()`	Return the current capacity of this `StringBuffer` object's array.
`char charAt(int index)`	Return the character located at `index` in this `StringBuffer` object's array. This method throws `IndexOutOfBoundsException` when `index` is negative or greater than or equal to this `StringBuffer` object's length.
`void ensureCapacity(int min)`	Ensure that this `StringBuffer` object's capacity is at least that specified by `min`. If the current capacity is less than `min`, a new internal array is created with greater capacity. The new capacity is set to the larger of `min` and the current capacity multiplied by 2, with 2 added to the result. No action is taken when `min` is negative or zero.
`int length()`	Return the number of characters stored in this `StringBuffer` object's array.
`StringBuffer reverse()`	Return this `StringBuffer` object with its array contents reversed.
`void setCharAt(int index, char ch)`	Replace the character at `index` with `ch`. This method throws `IndexOutOfBoundsException` when index is negative or greater than or equal to the length of this `StringBuffer` object's array.
`void setLength(int length)`	Set the length of this `StringBuffer` object's array to `length`. If the `length` argument is less than the current length, the array's contents are truncated. If the `length` argument is greater than or equal to the current length, sufficient null characters (`'\u0000'`) are appended to the array. This method throws `IndexOutOfBoundsException` when `length` is negative.
`String substring(int start)`	Return a new `String` object that contains all characters in this `StringBuffer` object's array starting with the character located at `start`. This method throws `IndexOutOfBoundsException` when `start` is less than 0 or greater than or equal to the length of this `StringBuffer` object's array.
`String toString()`	Return a new `String` object whose string equals the contents of this `StringBuffer` object's array.

A `StringBuffer` object's internal array is associated with the concepts of capacity and length. *Capacity* refers to the maximum number of characters that can be stored in the array before the array grows to accommodate additional characters. *Length* refers to the number of characters that are already stored in the array.

Chapter 6's Listing 6-13 declared a `toAlignedBinaryString()` method whose implementation included the following inefficient loop:

```
int numLeadingZeroes = 3;
```

```
String zeroesPrefix = "";
for (int j = 0; j < numLeadingZeroes; j++)
   zeroesPrefix += "0";
```

This loop is inefficient because each of the iterations creates a `StringBuffer` object and a `String` object. The compiler transforms this code fragment into the following fragment:

```
int numLeadingZeroes = 3;
String zeroesPrefix = "";
for (int j = 0; j < numLeadingZeroes; j++)
   zeroesPrefix = new StringBuffer().append(zeroesPrefix).append("0").toString();
```

NOTE: Starting with Java version 5, the compiler uses the more performant but otherwise identical `java.lang.StringBuilder` class instead of `StringBuffer`.

A more efficient way to code the previous loop involves creating a `StringBuffer` object prior to entering the loop, calling the appropriate `append()` method in the loop, and calling `toString()` after the loop. The following code fragment demonstrates this more efficient scenario:

```
int numLeadingZeroes = 3;
StringBuffer sb = new StringBuffer();
for (int j = 0; j < numLeadingZeroes; j++)
   sb.append('0');
String zeroesPrefix = sb.toString();
```

TIP: When performance matters, and where you are not using multiple threads (discussed later in this chapter), use the `StringBuilder` class instead of `StringBuffer`. `StringBuilder` provides the same methods as `StringBuffer`, but its methods are not synchronized (discussed later in this chapter). This lack of synchronization results in `StringBuilder` methods executing faster than their `StringBuffer` counterparts under most (if not all) Java implementations.

System

The `java.lang.System` class provides access to system-oriented resources, including standard input, standard output, and standard error.

`System` declares `in`, `out`, and `err` class fields that support standard input, standard output, and standard error, respectively. The first field is of type `java.io.InputStream`, and the last two fields are of type `java.io.PrintStream`. (I will formally introduce these classes in Chapter 10.)

`System` also declares a variety of utility methods, including those methods that are described in Table 7–4.

Table 7–4. *`System` Methods*

Method	Description
`static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)`	Copy the number of elements specified by `length` from the `src` array starting at zero-based offset `srcPos` into the `dest` array starting at zero-based offset `destPos`. This method throws `NullPointerException` when `src` or `dest` is `null`, `IndexOutOfBoundsException` when copying causes access to data outside array bounds, and `java.lang.ArrayStoreException` when an element in the `src` array could not be stored into the `dest` array because of a type mismatch.
`static long currentTimeMillis()`	Return the current system time in milliseconds since January 1, 1970 00:00:00 UTC.
`static void gc()`	Inform the virtual machine that now would be a good time to run the garbage collector. This is only a hint; there is no guarantee that the garbage collector will run.
`static String getProperty(String prop)`	Return the value of the *system property* (platform-specific attribute, such as a version number) identified by `prop` or null if such a property does not exist. System properties recognized by Android include `java.vendor.url`, `java.class.path`, `user.home`, `java.class.version`, `os.version`, `java.vendor`, `user.dir`, `user.timezone`, `path.separator`, `os.name`, `os.arch`, `line.separator`, `file.separator`, `user.name`, `java.version`, and `java.home`.
`static void runFinalization()`	Inform the virtual machine that now would be a good time to perform any outstanding object finalizations. This is only a hint; there is no guarantee that outstanding object finalizations will be performed.

Listing 7–2 demonstrates the `arraycopy()`, `currentTimeMillis()`, and `getProperty()` methods.

Listing 7–2. *Experimenting with `System` methods*

```
public class SystemTasks
{
   public static void main(String[] args)
   {
      int[] grades = { 86, 92, 78, 65, 52, 43, 72, 98, 81 };
      int[] gradesBackup = new int[grades.length];
      System.arraycopy(grades, 0, gradesBackup, 0, grades.length);
      for (int i = 0; i < gradesBackup.length; i++)
         System.out.println(gradesBackup[i]);
      System.out.println("Current time: " + System.currentTimeMillis());
      String[] propNames =
      {
         "java.vendor.url",
         "java.class.path",
         "user.home",
         "java.class.version",
         "os.version",
         "java.vendor",
         "user.dir",
         "user.timezone",
         "path.separator",
         "os.name",
         "os.arch",
         "line.separator",
         "file.separator",
         "user.name",
         "java.version",
         "java.home"
      };
      for (int i = 0; i < propNames.length; i++)
         System.out.println(propNames[i] + ": " +
                            System.getProperty(propNames[i]));
   }
}
```

Listing 7–2's main() method begins by demonstrating arraycopy(). It uses this method to copy the contents of a grades array to a gradesBackup array.

TIP: The arraycopy() method is the fastest portable way to copy one array to another. Also, when you write a class whose methods return a reference to an internal array, you should use arraycopy() to create a copy of the array, and then return the copy's reference. That way, you prevent clients from directly manipulating (and possibly screwing up) the internal array.

main() next calls currentTimeMillis() to return the current time as a milliseconds value. Because this value is not human-readable, you might want to use the java.util.Date class (discussed in Chapter 9). The Date() constructor calls currentTimeMillis() and its toString() method converts this value to a readable date and time.

main() concludes by demonstrating getProperty() in a for loop. This loop iterates over all of Table 7–4's property names, outputting each name and value.

When I run this application on my platform, it generates the following output:

```
86
92
78
65
52
43
72
98
81
Current time: 1274895119343
java.vendor.url: http://java.sun.com/
java.class.path: .
user.home: C:\Documents and Settings\Jeff Friesen
java.class.version: 50.0
os.version: 5.1
```

```
java.vendor: Sun Microsystems Inc.
user.dir: C:\prj\dev\ljfad\c06\code\SYSTEM~1
user.timezone:
path.separator: ;
os.name: Windows XP
os.arch: x86
line.separator:

file.separator: \
user.name: Jeff Friesen
java.version: 1.6.0_16
java.home: C:\Program Files\Java\jre6
```

NOTE: `line.separator` stores the actual line separator character/characters, not its/their representation (such as `\r\n`), which is why a blank line appears after `line.separator:`.

Threading API

Applications execute via *threads*, which are independent paths of execution through an application's code. When multiple threads are executing, each thread's path can differ from other thread paths. For example, a thread might execute one of a switch statement's cases, and another thread might execute another of this statement's cases.

NOTE: Applications use threads to improve performance. Some applications can get by with only the default main thread to carry out their tasks, but other applications need additional threads to perform time-intensive tasks in the background, so that they remain responsive to their users.

The virtual machine gives each thread its own method-call stack to prevent threads from interfering with each other. Separate stacks let threads keep track of their next instructions to execute, which can differ from thread to thread. The stack also provides a thread with its own copy of method parameters, local variables, and return value.

Java supports threads via its Threading API. This API consists of one interface (`Runnable`) and four classes (`Thread`, `ThreadGroup`, `ThreadLocal`, and `InheritableThreadLocal`) in the `java.lang` package. After exploring `Runnable` and `Thread` (and mentioning `ThreadGroup` during this exploration), this section explores thread synchronization, `ThreadLocal`, and `InheritableThreadLocal`.

NOTE: Java version 5 introduced the `java.util.concurrent` package as a high-level alternative to the low-level Threading API. (I will discuss this package in Chapter 9.) Although `java.util.concurrent` is the preferred API for working with threads, you should also be somewhat familiar with Threading because it is helpful in simple threading scenarios. Also, you might have to analyze someone else's source code that depends on Threading.

Runnable and Thread

Java provides the `Runnable` interface to identify those objects that supply code for threads to execute via this interface's solitary `public void run()` method—a thread receives no arguments and returns no value. Classes implement `Runnable` to supply this code, and one of these classes is `Thread`.

`Thread` provides a consistent interface to the underlying operating system's threading architecture. (The operating system is typically responsible for creating and managing threads.) `Thread` makes it possible to associate code with threads, as well as start and manage those threads. Each `Thread` instance associates with a single thread.

`Thread` declares several constructors for initializing `Thread` objects. Some of these constructors take `Runnable` arguments: you can supply code to run without having to extend `Thread`. Other constructors do not take `Runnable` arguments: you must extend `Thread` and override its `run()` method to supply the code to run.

For example, `Thread(Runnable runnable)` initializes a new `Thread` object to the specified `runnable` whose code is to be executed. In contrast, `Thread()` does not initialize `Thread` to a `Runnable` argument. Instead, your `Thread` subclass provides a constructor that calls `Thread()`, and the subclass also overrides `Thread`'s `run()` method.

In the absence of an explicit name argument, each constructor assigns a unique default name (starting with `Thread-`) to the `Thread` object. Names make it possible to differentiate threads. In contrast to the previous two constructors, which choose default names, `Thread(String threadName)` lets you specify your own thread name.

`Thread` also declares methods for starting and managing threads. Table 7–5 describes many of the more useful methods.

Table 7–5. *`Thread` Methods*

Method	Description
`static Thread currentThread()`	Return the `Thread` object associated with the thread that calls this method.
`String getName()`	Return the name associated with this `Thread` object.
`Thread.State getState()`	Return the state of the thread associated with this `Thread` object. The state is identified by the `Thread.State` enum as one of `BLOCKED` (waiting to acquire a lock, discussed later), `NEW` (created but not started), `RUNNABLE` (executing), `TERMINATED` (the thread has died), `TIMED_WAITING` (waiting for a specified amount of time to elapse), or `WAITING` (waiting indefinitely).
`void interrupt()`	Set the interrupt status flag in this `Thread` object. If the associated thread is blocked or waiting, clear this flag and wake up the thread by throwing an instance of the checked `InterruptedException` class.

Method	Description
`static boolean interrupted()`	Return true when the thread associated with this `Thread` object has a pending interrupt request. Clear the interrupt status flag.
`boolean isAlive()`	Return true to indicate that this `Thread` object's associated thread is alive and not dead. A thread's lifespan ranges from just before it is actually started within the `start()` method to just after it leaves the `run()` method, at which point it dies.
`boolean isDaemon()`	Return true when the thread associated with this `Thread` object is a *daemon thread*, a thread that acts as a helper to a *user thread* (nondaemon thread) and dies automatically when the application's last nondaemon thread dies so the application can exit.
`boolean isInterrupted()`	Return true when the thread associated with this `Thread` object has a pending interrupt request.
`void join()`	The thread that calls this method on this `Thread` object waits for the thread associated with this object to die. This method throws `InterruptedException` when this `Thread` object's `interrupt()` method is called.
`void join(long millis)`	The thread that calls this method on this `Thread` object waits for the thread associated with this object to die, or until `millis` milliseconds have elapsed, whichever happens first. This method throws `InterruptedException` when this `Thread` object's `interrupt()` method is called.
`void setDaemon(boolean isDaemon)`	Mark this `Thread` object's associated thread as a daemon thread when `isDaemon` is true. This method throws `java.lang.IllegalThreadStateException` when the thread has not yet been created and started.
`void setName(String threadName)`	Assign `threadName`'s value to this `Thread` object as the name of its associated thread.
`static void sleep(long time)`	Pause the thread associated with this `Thread` object for `time` milliseconds. This method throws `InterruptedException` when this `Thread` object's `interrupt()` method is called while the thread is sleeping.
`void start()`	Create and start this `Thread` object's associated thread. This method throws `IllegalThreadStateException` when the thread was previously started and is running or has died.

Listing 7–3 introduces you to the Threading API via a `main()` method that demonstrates `Runnable`, `Thread(Runnable runnable)`, `currentThread()`, `getName()`, and `start()`.

Listing 7–3. *A pair of counting threads*

```
public class CountingThreads
{
   public static void main(String[] args)
   {
      Runnable r = new Runnable()
                   {
                      @Override
                      public void run()
                      {
                         String name = Thread.currentThread().getName();
                         int count = 0;
                         while (true)
                            System.out.println(name + ": " + count++);
                      }
                   };
      Thread thdA = new Thread(r);
      Thread thdB = new Thread(r);
      thdA.start();
      thdB.start();
   }
}
```

The default main thread that executes `main()` first instantiates an anonymous class that implements `Runnable`. It then creates two `Thread` objects, initializing each object to the runnable, and calls `Thread`'s `start()` method to create and start both threads. After completing these tasks, the main thread exits `main()` and dies.

Each of the two started threads executes the runnable's `run()` method. It calls `Thread`'s `currentThread()` method to obtain its associated `Thread` instance, uses this instance to call `Thread`'s `getName()` method to return its name, initializes `count` to 0, and enters an infinite loop where it outputs `name` and `count` and increments `count` on each iteration.

TIP: To stop an application that does not end, press the Ctrl and C keys simultaneously.

I observe both threads alternating in their execution when I run this application on the Windows XP platform. Partial output from one run appears here:

```
Thread-0: 0
Thread-0: 1
Thread-0: 2
Thread-0: 3
Thread-0: 4
Thread-0: 5
Thread-0: 6
Thread-0: 7
Thread-1: 0
Thread-1: 1
Thread-1: 2
Thread-1: 3
Thread-1: 4
Thread-1: 5
Thread-1: 6
```

```
Thread-1: 7
Thread-1: 8
Thread-1: 9
Thread-1: 10
Thread-1: 11
Thread-1: 12
Thread-1: 13
Thread-1: 14
Thread-1: 15
Thread-0: 8
Thread-0: 9
```

When a computer has enough processors and/or processor cores, the computer's operating system assigns a separate thread to each processor or core so the threads execute *concurrently* (at the same time). When a computer does not have enough processors and/or cores, a thread must wait its turn to use the shared processor/core.

The operating system uses a *scheduler* (`http://en.wikipedia.org/wiki/Scheduling_%28computing%29`) to determine when a waiting thread executes. The following list identifies three different schedulers:

- Linux 2.6 through 2.6.22 uses the O(1) scheduler (`http://en.wikipedia.org/wiki/O%281%29_scheduler`).
- Linux 2.6.23 uses the Completely Fair Scheduler (`http://en.wikipedia.org/wiki/Completely_Fair_Scheduler`).
- Windows NT-based operating systems (NT, XP, and Vista) use a multilevel feedback queue scheduler (`http://en.wikipedia.org/wiki/Multilevel_feedback_queue`).

The previous output from the counting threads application resulted from running this application via Windows XP's *multilevel feedback queue* scheduler. Because of this scheduler, both threads take turns executing.

> **CAUTION:** Although this output indicates that the first thread starts executing, never assume that the thread associated with the `Thread` object whose `start()` method is called first is the first thread to execute. While this might be true of some schedulers, it might not be true of others.

A multilevel feedback queue and many other thread schedulers take the concept of *priority* (thread relative importance) into account. They often combine *preemptive scheduling* (higher priority threads *preempt*—interrupt and run instead of—lower priority threads) with *round robin scheduling* (equal priority threads are given equal slices of time, which are known as *time slices*, and take turns executing).

`Thread` supports priority via its `void setPriority(int priority)` method (set the priority of this `Thread` object's thread to `priority`, which ranges from `Thread.MIN_PRIORITY` to `Thread.MAX_PRIORITY`—`Thread.NORMAL_PRIORITY` identifies the default priority) and `int getPriority()` method (return the current priority).

CAUTION: Using the `setPriority()` method can impact an application's portability across platforms because different schedulers can handle a priority change in different ways. For example, one platform's scheduler might delay lower priority threads from executing until higher priority threads finish. This delaying can lead to *indefinite postponement* or *starvation* because lower priority threads "starve" while waiting indefinitely for their turn to execute, and this can seriously hurt the application's performance. Another platform's scheduler might not indefinitely delay lower priority threads, improving application performance.

Listing 7–4 refactors Listing 7–3's `main()` method to give each thread a nondefault name, and to put each thread to sleep after outputting `name` and `count`.

Listing 7–4. *A pair of counting threads revisited*

```
public static void main(String[] args)
{
   Runnable r = new Runnable()
                {
                   @Override
                   public void run()
                   {
                      String name = Thread.currentThread().getName();
                      int count = 0;
                      while (true)
                      {
                         System.out.println(name + ": " + count++);
                         try
                         {
                            Thread.sleep(100);
                         }
                         catch (InterruptedException ie)
                         {
                         }
                      }
                   }
                };
   Thread thdA = new Thread(r);
   thdA.setName("A");
   Thread thdB = new Thread(r);
   thdB.setName("B");
   thdA.start();
   thdB.start();
}
```

Threads A and B execute `Thread.sleep(100);` to sleep for 100 milliseconds. This sleep results in each thread executing more frequently, as the following partial output reveals:

```
A: 0
B: 0
A: 1
B: 1
A: 2
B: 2
A: 3
```

```
B: 3
A: 4
B: 4
A: 5
B: 5
```

A thread will occasionally start another thread to perform a lengthy calculation, download a large file, or perform some other time-consuming activity. After finishing its other tasks, the thread that started the worker thread is ready to process the results of the worker thread and waits for the worker thread to finish and die.

It is possible to wait for the worker thread to die by using a while loop that repeatedly calls `Thread`'s `isAlive()` method on the worker thread's `Thread` object and sleeps for a certain length of time when this method returns true. However, Listing 7–5 demonstrates a less verbose alternative: the `join()` method.

Listing 7–5. *Joining the default main thread with a background thread*

```
public static void main(String[] args)
{
   Runnable r = new Runnable()
                {
                   @Override
                   public void run()
                   {
                      System.out.println("Worker thread is simulating " +
                                         "work by sleeping for 5 seconds.");
                      try
                      {
                         Thread.sleep(5000);
                      }
                      catch (InterruptedException ie)
                      {
                      }
                      System.out.println("Worker thread is dying");
                   }
                };
   Thread thd = new Thread(r);
   thd.start();
   System.out.println("Default main thread is doing work.");
   try
   {
      Thread.sleep(2000);
   }
   catch (InterruptedException ie)
   {
   }
   System.out.println("Default main thread has finished its work.");
   System.out.println("Default main thread is waiting for worker thread " +
                      "to die.");
   try
   {
      thd.join();
   }
   catch (InterruptedException ie)
   {
   }
   System.out.println("Main thread is dying");
}
```

This listing demonstrates the default main thread starting a worker thread, performing some work, and then waiting for the worker thread to die by calling `join()` via the worker thread's `thd` object. When you run this application, you will discover output similar to the following (message order might differ somewhat):

```
Default main thread is doing work.
Worker thread is simulating work by sleeping for 5 seconds.
Default main thread has finished its work.
Default main thread is waiting for worker thread to die.
Worker thread is dying
Main thread is dying
```

Every `Thread` object belongs to some `ThreadGroup` object; `Thread` declares a `ThreadGroup getThreadGroup()` method that returns this object. You should ignore thread groups because they are not that useful. If you need to logically group `Thread` objects, you should use an array or collection instead.

CAUTION: Various `ThreadGroup` methods are flawed. For example, `int enumerate(Thread[] threads)` will not include all active threads in its enumeration when its `threads` array argument is too small to store their `Thread` objects. Although you might think that you could use the return value from the `int activeCount()` method to properly size this array, there is no guarantee that the array will be large enough because `activeCount()`'s return value fluctuates with the creation and death of threads.

However, you should still know about `ThreadGroup` because of its contribution in handling exceptions that are thrown while a thread is executing. Listing 7–6 sets the stage for learning about exception handling by presenting a `run()` method that attempts to divide an integer by 0, which results in a thrown `ArithmeticException` instance.

Listing 7–6. *Throwing an exception from the `run()` method*

```
public static void main(String[] args)
{
   Runnable r = new Runnable()
                {
                   @Override
                   public void run()
                   {
                      int x = 1/0; // Line 8
                   }
                };
   Thread thd = new Thread(r);
   thd.start();
}
```

Run this application and you will see an exception trace that identifies the thrown `ArithmeticException`:

```
Exception in thread "Thread-0" java.lang.ArithmeticException: / by zero
```

```
	at ExceptionThread$1.run(ExceptionThread.java:8)
	at java.lang.Thread.run(Unknown Source)
```

When an exception is thrown out of the run() method, the thread terminates and the following activities take place:

- The virtual machine looks for an instance of Thread.UncaughtExceptionHandler installed via Thread's void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh) method. When this handler is found, it passes execution to the instance's void uncaughtException(Thread t, Throwable e) method, where t identifies the Thread object of the thread that threw the exception, and e identifies the thrown exception or error—perhaps an OutOfMemoryError instance was thrown. If this method throws an exception/error, the exception/error is ignored by the virtual machine.
- Assuming that setUncaughtExceptionHandler() was not called to install a handler, the virtual machine passes control to the associated ThreadGroup object's uncaughtException(Thread t, Throwable e) method. Assuming that ThreadGroup was not extended, and that its uncaughtException() method was not overridden to handle the exception, uncaughtException() passes control to the parent ThreadGroup object's uncaughtException() method when a parent ThreadGroup is present. Otherwise, it checks to see if a default uncaught exception handler has been installed (via Thread's static void setDefaultUncaughtExceptionHandler (Thread.UncaughtExceptionHandler handler) method.) If a default uncaught exception handler has been installed, its uncaughtException() method is called with the same two arguments. Otherwise, uncaughtException() checks its Throwable argument to determine if it is an instance of java.lang.ThreadDeath. If so, nothing special is done. Otherwise, as Listing 7–6's exception message shows, a message containing the thread's name, as returned from the thread's getName() method, and a stack backtrace, using the Throwable argument's printStackTrace() method, is printed to the standard error stream.

Listing 7–7 demonstrates Thread's setUncaughtExceptionHandler() and setDefaultUncaughtExceptionHandler() methods.

Listing 7–7. ***Demonstrating uncaught exception handlers***

```
public static void main(String[] args)
{
   Runnable r = new Runnable()
                {
                   @Override
                   public void run()
                   {
                      int x = 1/0;
                   }
                };
   Thread thd = new Thread(r);
   Thread.UncaughtExceptionHandler uceh;
   uceh = new Thread.UncaughtExceptionHandler()
          {
             public void uncaughtException(Thread t, Throwable e)
             {
                System.out.println("Caught throwable " + e + " for thread "
                                   + t);
             }
          };
   thd.setUncaughtExceptionHandler(uceh);
   uceh = new Thread.UncaughtExceptionHandler()
          {
             public void uncaughtException(Thread t, Throwable e)
             {
                System.out.println("Default uncaught exception handler");
                System.out.println("Caught throwable " + e + " for thread "
                                   + t);
             }
          };
   thd.setDefaultUncaughtExceptionHandler(uceh);
   thd.start();
}
```

When you run this application, you will observe the following output:

```
Caught throwable java.lang.ArithmeticException: / by zero for thread↵
 Thread[Thread-0,5,main]
```

You will not also see the default uncaught exception handler's output because the default handler is not called. To see that output, you must comment out `thd.setUncaughtExceptionHandler(uceh);`. If you also comment out `thd.setDefaultUncaughtExceptionHandler(uceh);`, you will see Listing 7–6's output.

CAUTION: `Thread` declares several deprecated methods, including `stop()` (stop an executing thread). These methods have been deprecated because they are unsafe. Do *not* use these deprecated methods. (I will show you how to safely stop a thread later in this chapter.)

Also, you should avoid the `static void yield()` method, which is intended to switch execution from the current thread to another thread, because it can affect portability and hurt application performance. Although `yield()` might switch to another thread on some platforms (which can improve performance), `yield()` might only return to the current thread on other platforms (which hurts performance because the `yield()` call has only wasted time).

Thread Synchronization

Throughout its execution, each thread is isolated from other threads because it has been given its own method-call stack. However, threads can still interfere with each other when they access and manipulate shared data. This interference can corrupt the shared data, and this corruption can cause an application to fail.

For example, consider a checking account in which a husband and wife have joint access. Suppose that the husband and wife decide to empty this account at the same time without knowing that the other is doing the same thing. Listing 7–8 demonstrates this scenario.

Listing 7–8. *A problematic checking account*

```
public class CheckingAccount
{
   private int balance;
   public CheckingAccount(int initialBalance)
   {
      balance = initialBalance;
   }
   public boolean withdraw(int amount)
   {
      if (amount <= balance)
      {
         try
         {
            Thread.sleep((int)(Math.random()*200));
         }
         catch (InterruptedException ie)
         {
         }
         balance -= amount;
         return true;
      }
      return false;
   }
   public static void main(String[] args)
   {
      final CheckingAccount ca = new CheckingAccount(100);
      Runnable r = new Runnable()
                   {
                      public void run()
                      {
                         String name = Thread.currentThread().getName();
                         for (int i = 0; i < 10; i++)
                             System.out.println (name + " withdraws $10: " +
                                                 ca.withdraw(10));
                      }
                   };
      Thread thdHusband = new Thread(r);
      thdHusband.setName("Husband");
      Thread thdWife = new Thread(r);
      thdWife.setName("Wife");
      thdHusband.start();
      thdWife.start();
   }
}
```

This application lets more money be withdrawn than is available in the account. For example, the following output reveals $110 being withdrawn when only $100 is available:

```
Wife withdraws $10: true
Wife withdraws $10: true
Husband withdraws $10: true
Wife withdraws $10: true
Husband withdraws $10: true
Wife withdraws $10: true
Husband withdraws $10: true
Husband withdraws $10: true
Husband withdraws $10: true
Husband withdraws $10: true
Husband withdraws $10: false
Husband withdraws $10: false
Husband withdraws $10: false
Husband withdraws $10: false
Wife withdraws $10: true
Wife withdraws $10: false
Wife withdraws $10: false
Wife withdraws $10: false
Wife withdraws $10: false
Wife withdraws $10: false
```

The reason why more money is withdrawn than is available for withdrawal is that a race condition exists between the husband and wife threads.

NOTE: A *race condition* is a scenario in which multiple threads update the same object at the same time or nearly at the same time. Part of the object stores values written to it by one thread, and another part of the object stores values written to it by another thread.

The race condition exists because the actions of checking the amount for withdrawal to ensure that it is less than what appears in the balance and deducting the amount from the balance are not *atomic* (indivisible) operations. (Although atoms are divisible, *atomic* is commonly used to refer to something being indivisible.)

NOTE: The `Thread.sleep()` method call that sleeps for a variable amount of time (up to a maximum of 199 milliseconds) is present so that you can observe more money being withdrawn than is available for withdrawal. Without this method call, you might have to execute the application hundreds of times (or more) to witness this problem, because the scheduler might rarely pause a thread between the `amount <= balance` expression and the `balance -= amount;` expression statement—the code executes rapidly.

Consider the following scenario:

- The Husband thread executes `withdraw()`'s `amount <= balance` expression, which returns true. The scheduler pauses the Husband thread and lets the Wife thread execute.
- The Wife thread executes `withdraw()`'s `amount <= balance` expression, which returns true.

- The Wife thread performs the withdrawal. The scheduler pauses the Wife thread and lets the Husband thread execute.
- The Husband thread performs the withdrawal.

This problem can be corrected by synchronizing access to `withdraw()` so that only one thread at a time can execute inside this method. You synchronize access at the method level by adding reserved word `synchronized` to the method header prior to the method's return type; for example, `public synchronized boolean withdraw(int amount)`.

As I demonstrate later, you can also synchronize access to a block of statements by specifying `synchronized(`*`object`*`) { /* synchronized statements */ }`, where *`object`* is an arbitrary object reference. No thread can enter a synchronized method or block until execution leaves the method/block; this is known as *mutual exclusion*.

Synchronization is implemented in terms of monitors and locks. A *monitor* is a concurrency construct for controlling access to a *critical section*, a region of code that must execute atomically. It is identified at the source code level as a synchronized method or a synchronized block.

A *lock* is a token that a thread must acquire before a monitor allows that thread to execute inside a monitor's critical section. The token is released automatically when the thread exits the monitor, to give another thread an opportunity to acquire the token and enter the monitor.

NOTE: A thread that has acquired a lock does not release this lock when it calls one of `Thread`'s `sleep()` methods.

A thread entering a synchronized instance method acquires the lock associated with the object on which the method is called. A thread entering a synchronized class method acquires the lock associated with the class's `Class` object. Finally, a thread entering a synchronized block acquires the lock associated with the block's controlling object.

TIP: `Thread` declares a `public static boolean holdsLock(Object o)` method that returns true when the calling thread holds the monitor lock on object `o`. You will find this method handy in assertion statements, such as `assert Thread.holdsLock(o);`.

The need for synchronization is often subtle. For example, Listing 7–9's `ID` utility class declares a `getNextID()` method that returns a unique `long`-based ID, perhaps to be used when generating unique filenames. Although you might not think so, this method can cause data corruption and return duplicate values.

Listing 7–9. *A utility class for returning unique IDs*

```
public class ID
{
   private static long nextID = 0;
   public static long getNextID()
   {
      return nextID++;
   }
}
```

There are two lack-of-synchronization problems with `getNextID()`. Because 32-bit virtual machine implementations require two steps to update a 64-bit long integer, adding 1 to `nextID` is not atomic: the scheduler could interrupt a thread that has only updated half of `nextID`, which corrupts the contents of this variable.

NOTE: Variables of type `long` and `double` are subject to corruption when being written to in an unsynchronized context on 32-bit virtual machines. This problem does not occur with variables of type `boolean`, `byte`, `char`, `float`, `int`, or `short`; each type occupies 32 bits or less.

Assume that multiple threads call `getNextID()`. Because postincrement (++) reads and writes the `nextID` field in two steps, multiple threads might retrieve the same value. For example, thread A executes ++, reading `nextID` but not incrementing its value before being interrupted by the scheduler. Thread B now executes and reads the same value.

Both problems can be corrected by synchronizing access to `nextID` so that only one thread can execute this method's code. All that is required is to add `synchronized` to the method header prior to the method's return type; for example, `public static synchronized int getNextID()`.

Synchronization is also used to communicate between threads. For example, you might design your own mechanism for stopping a thread (because you cannot use `Thread`'s unsafe `stop()` methods for this task). Listing 7–10 shows how you might accomplish this task.

Listing 7–10. *Attempting to stop a thread*

```
public static void main(String[] args)
{
   class StoppableThread extends Thread
   {
      private boolean stopped = false;
      @Override
      public void run()
      {
         while(!stopped)
           System.out.println("running");
      }
      public void stopThread()
      {
         stopped = true;
      }
   }
   StoppableThread thd = new StoppableThread();
   thd.start();
   try
   {
      Thread.sleep(1000); // sleep for 1 second
   }
   catch (InterruptedException ie)
   {
   }
   thd.stopThread();
}
```

Listing 7–10 introduces a `main()` method with a local class named `StoppableThread` that subclasses `Thread`. `StoppableThread` declares a `stopped` field initialized to `false`, a `stopThread()` method that sets this field to `true`, and a `run()` method whose infinite loop checks `stopped` on each loop iteration to see if its value has changed to `true`.

After instantiating `StoppableThread`, the default main thread starts the thread associated with this `Thread` object. It then sleeps for one second and calls `StoppableThread`'s `stop()` method before dying. When you run this application on a single-processor/single-core machine, you will probably observe the application stopping.

You might not see this stoppage when the application runs on a multiprocessor machine or a uniprocessor machine with multiple cores. For performance reasons, each processor or core probably has its own cache with its own copy of `stopped`. When one thread modifies its copy of this field, the other thread's copy of `stopped` is not changed.

Listing 7–11 refactors Listing 7–10 to guarantee that the application will run correctly on all kinds of machines.

Listing 7–11. *Guaranteed stoppage on a multiprocessor/multicore machine*

```
public static void main(String[] args)
{
   class StoppableThread extends Thread
   {
      private boolean stopped = false;
      @Override
      public void run()
      {
         while(!isStopped())
           System.out.println("running");
      }
      public synchronized void stopThread()
      {
         stopped = true;
      }
      private synchronized boolean isStopped()
      {
         return stopped;
      }
   }
   StoppableThread thd = new StoppableThread();
   thd.start();
   try
   {
      Thread.sleep(1000); // sleep for 1 second
   }
   catch (InterruptedException ie)
   {
   }
   thd.stopThread();
}
```

Listing 7–11's `stopThread()` and `isStopped()` methods are synchronized to support thread communication (between the default main thread that calls `stopThread()` and the started thread that executes inside `run()`). When a thread enters one of these methods, it is guaranteed to access a single shared copy of the `stopped` field (not a cached copy).

Synchronization is necessary to support mutual exclusion or mutual exclusion combined with thread communication. However, there exists an alternative to synchronization when the only purpose is to communicate between threads. This alternative is reserved word `volatile`, which Listing 7–12 demonstrates.

Listing 7–12. *The `volatile` alternative to synchronization for thread communication*

```
public static void main(String[] args)
{
   class StoppableThread extends Thread
   {
      private volatile boolean stopped = false;
      @Override
      public void run()
      {
         while(!stopped)
           System.out.println("running");
      }
      public void stopThread()
      {
         stopped = true;
      }
   }
   StoppableThread thd = new StoppableThread();
   thd.start();
   try
   {
      Thread.sleep(1000); // sleep for 1 second
   }
   catch (InterruptedException ie)
   {
   }
   thd.stopThread();
}
```

Listing 7–12 declares `stopped` to be `volatile`; threads that access this field will always access a single shared copy (not cached copies on multiprocessor/multicore machines). In addition to generating code that is less verbose, `volatile` might offer improved performance over synchronization.

CAUTION: You should only use `volatile` in the context of thread communication. Also, you can only use this reserved word in the context of field declarations. Although you can declare `double` and `long` fields `volatile`, you should avoid doing so on 32-bit virtual machines because it takes two operations to access a `double` or `long` variable's value, and mutual exclusion via synchronization is required to access their values safely.

`Object`'s `wait()`, `notify()`, and `notifyAll()` methods support a form of thread communication where a thread voluntarily waits for some *condition* (a prerequisite for continued execution) to arise, at which time another thread notifies the waiting thread that it can continue. `wait()` causes its calling thread to wait on an object's monitor, and `notify()` and `notifyAll()` wake up one or all threads waiting on the monitor.

CAUTION: Because the `wait()`, `notify()`, and `notifyAll()` methods depend on a lock, they cannot be called from outside of a synchronized method or synchronized block. If you fail to heed this warning, you will encounter a thrown instance of the `java.lang.IllegalMonitorStateException` class. Also, a thread that has acquired a lock releases this lock when it calls one of `Object`'s `wait()` methods.

A classic example of thread communication involving conditions is the relationship between a producer thread and a consumer thread. The producer thread produces data items to be consumed by the consumer thread. Each produced data item is stored in a shared variable.

Imagine that the threads are not communicating and are running at different speeds. The producer might produce a new data item and record it in the shared variable before the consumer retrieves the previous data item for processing. Also, the consumer might retrieve the contents of the shared variable before a new data item is produced.

To overcome those problems, the producer thread must wait until it is notified that the previously produced data item has been consumed, and the consumer thread must wait until it is notified that a new data item has been produced. Listing 7–13 shows you how to accomplish this task via `wait()` and `notify()`.

Listing 7–13. *The producer-consumer relationship*

```
public class PC
{
   public static void main(String[] args)
   {
      Shared s = new Shared();
      new Producer(s).start();
      new Consumer(s).start();
   }
}
class Shared
{
   private char c = '\u0000';
   private boolean writeable = true;
   synchronized void setSharedChar(char c)
   {
      while (!writeable)
         try
         {
            wait();
         }
         catch (InterruptedException e) {}
      this.c = c;
      writeable = false;
      notify();
   }
   synchronized char getSharedChar()
   {
      while (writeable)
         try
         {
            wait();
         }
         catch (InterruptedException e) {}
      writeable = true;
      notify();
      return c;
   }
}
class Producer extends Thread
{
   private Shared s;
   Producer(Shared s)
   {
      this.s = s;
   }
   @Override
   public void run()
   {
      for (char ch = 'A'; ch <= 'Z'; ch++)
      {
         synchronized(s)
         {
            s.setSharedChar(ch);
            System.out.println(ch + " produced by producer.");
         }
      }
   }
}
class Consumer extends Thread
{
   private Shared s;
   Consumer(Shared s)
   {
      this.s = s;
   }
   @Override
   public void run()
   {
      char ch;
```

```
         do
         {
            synchronized(s)
            {
               ch = s.getSharedChar();
               System.out.println(ch + " consumed by consumer.");
            }
         }
         while (ch != 'Z');
      }
   }
```

The application creates a `Shared` object and two threads that get a copy of the object's reference. The producer calls the object's `setSharedChar()` method to save each of 26 uppercase letters; the consumer calls the object's `getSharedChar()` method to acquire each letter.

The `writeable` instance field tracks two conditions: the producer waiting on the consumer to consume a data item, and the consumer waiting on the producer to produce a new data item. It helps coordinate execution of the producer and consumer. The following scenario, where the consumer executes first, illustrates this coordination:

1. The consumer executes `s.getSharedChar()` to retrieve a letter.
2. Inside of that synchronized method, the consumer calls `wait()` because `writeable` contains true. The consumer now waits until it receives notification from the producer.
3. The producer eventually executes `s.setSharedChar(ch);`.
4. When the producer enters that synchronized method (which is possible because the consumer released the lock inside of the `wait()` method prior to waiting), the producer discovers `writeable`'s value to be true and does not call `wait()`.
5. The producer saves the character, sets `writeable` to `false` (which will cause the producer to wait on the next `setSharedChar()` call when the consumer has not consumed the character by that time), and calls `notify()` to awaken the consumer (assuming the consumer is waiting).
6. The producer exits `setSharedChar(char c)`.
7. The consumer wakes up (and reacquires the lock), sets `writeable` to `true` (which will cause the consumer to wait on the next `getSharedChar()` call when the producer has not produced a character by that time), notifies the producer to awaken that thread (assuming the producer is waiting), and returns the shared character.

Although the synchronization works correctly, you might observe output (on some platforms) that shows multiple producing messages before a consuming message. For example, you might see `A produced by producer.`, followed by `B produced by` producer., followed by A consumed by consumer., at the beginning of the application's output.

This strange output order is caused by the call to setSharedChar() followed by its companion System.out.println() method call not being atomic, and by the call to getSharedChar() followed by its companion System.out.println() method call not being atomic. The output order is corrected by wrapping each of these method call pairs in a synchronized block that synchronizes on the Shared object referenced by s.

When you run this application, its output should always appear in the same alternating order, as shown next (only the first few lines are shown for brevity):

```
A produced by producer.
A consumed by consumer.
B produced by producer.
B consumed by consumer.
C produced by producer.
C consumed by consumer.
D produced by producer.
D consumed by consumer.
```

CAUTION: Never call wait() outside of a loop. The loop tests the condition (!writeable or writeable in the previous example) before and after the wait() call. Testing the condition before calling wait() ensures *liveness*. If this test was not present, and if the condition held and notify() had been called prior to wait() being called, it is unlikely that the waiting thread would ever wake up. Retesting the condition after calling wait() ensures *safety*. If retesting did not occur, and if the condition did not hold after the thread had awakened from the wait() call (perhaps another thread called notify() accidentally when the condition did not hold), the thread would proceed to destroy the lock's protected invariants.

Too much synchronization can be problematic. If you are not careful, you might encounter a situation where locks are acquired by multiple threads, neither thread holds its own lock but holds the lock needed by some other thread, and neither thread can enter and later exit its critical section to release its held lock because some other thread holds the lock to that critical section. Listing 7–14's atypical example demonstrates this scenario, which is known as *deadlock*.

Listing 7–14. *A pathological case of deadlock*

```
public class Deadlock
{
   private Object lock1 = new Object();
   private Object lock2 = new Object();
   public void instanceMethod1()
   {
      synchronized(lock1)
      {
         synchronized(lock2)
         {
            System.out.println("first thread in instanceMethod1");
            // critical section guarded first by
            // lock1 and then by lock2
         }
      }
   }
   public void instanceMethod2()
   {
      synchronized(lock2)
      {
         synchronized(lock1)
         {
            System.out.println("second thread in instanceMethod2");
            // critical section guarded first by
            // lock2 and then by lock1
         }
      }
   }
   public static void main(String[] args)
   {
      final Deadlock dl = new Deadlock();
      Runnable r1 = new Runnable()
                    {
                       @Override
                       public void run()
                       {
                          while(true)
                             dl.instanceMethod1();
                       }
                    };
      Thread thdA = new Thread(r1);
      Runnable r2 = new Runnable()
                    {
                       @Override
                       public void run()
                       {
                          while(true)
                             dl.instanceMethod2();
                       }
                    };
      Thread thdB = new Thread(r2);
      thdA.start();
      thdB.start();
   }
}
```

Listing 7–14's thread A and thread B call `instanceMethod1()` and `instanceMethod2()`, respectively, at different times. Consider the following execution sequence:

1. Thread A calls `instanceMethod1()`, obtains the lock assigned to the `lock1`-referenced object, and enters its outer critical section (but has not yet acquired the lock assigned to the `lock2`-referenced object).

2. Thread B calls `instanceMethod2()`, obtains the lock assigned to the `lock2`-referenced object, and enters its outer critical section (but has not yet acquired the lock assigned to the `lock1`-referenced object).

3. Thread A attempts to acquire the lock associated with `lock2`. The virtual machine forces the thread to wait outside of the inner critical section because thread B holds that lock.
4. Thread B attempts to acquire the lock associated with `lock1`. The virtual machine forces the thread to wait outside of the inner critical section because thread A holds that lock.
5. Neither thread can proceed because the other thread holds the needed lock. We have a deadlock situation and the program (at least in the context of the two threads) freezes up.

Although the previous example clearly identifies a deadlock state, it is often not that easy to detect deadlock. For example, your code might contain the following circular relationship among various classes (in several source files):

- Class A's synchronized method calls class B's synchronized method.
- Class B's synchronized method calls class C's synchronized method.
- Class C's synchronized method calls class A's synchronized method.

If thread A calls class A's synchronized method and thread B calls class C's synchronized method, thread B will block when it attempts to call class A's synchronized method and thread A is still inside of that method. Thread A will continue to execute until it calls class C's synchronized method, and then block. Deadlock results.

NOTE: Neither the Java language nor the virtual machine provides a way to prevent deadlock, and so the burden falls on you. The simplest way to prevent deadlock from happening is to avoid having either a synchronized method or a synchronized block call another synchronized method/block. Although this advice prevents deadlock from happening, it is impractical because one of your synchronized methods/blocks might need to call a synchronized method in a Java API, and the advice is overkill because the synchronized method/block being called might not call any other synchronized method/block, so deadlock would not occur.

You will sometimes want to associate per-thread data (such a user ID) with a thread. Although you can accomplish this task with a local variable, you can only do so while the local variable exists. You could use an instance field to keep this data around longer, but then you would have to deal with synchronization. Thankfully, Java supplies `ThreadLocal` as a simple alternative.

Each instance of the `ThreadLocal` class describes a *thread-local variable*, which is a variable that provides a separate storage slot to each thread that accesses the variable. You can think of a thread-local variable as a multislot variable in which each thread can store a different value in the same variable. Each thread sees only its value and is unaware of other threads having their own values in this variable.

ThreadLocal is generically declared as ThreadLocal<T>, where T identifies the type of value that is stored in the variable. This class declares the following constructor and methods:

- ThreadLocal() creates a new thread-local variable.
- T get() returns the value in the calling thread's storage slot. If an entry does not exist when the thread calls this method, get() calls initialValue().
- T initialValue() creates the calling thread's storage slot and stores an initial (default) value in this slot. The initial value defaults to null. You must subclass ThreadLocal and override this protected method to provide a more suitable initial value.
- void remove() removes the calling thread's storage slot. If this method is followed by get() with no intervening set(), get() calls initialValue().
- void set(T value) sets the value of the calling thread's storage slot to value.

Listing 7–15 shows you how to use ThreadLocal to associate a different user ID with each of two threads.

Listing 7–15. *Different user IDs for different threads*

```
private static volatile ThreadLocal<String> userID =
   new ThreadLocal<String>();
public static void main(String[] args)
{
   Runnable r = new Runnable()
                {
                   @Override
                   public void run()
                   {
                      String name = Thread.currentThread().getName();
                      if (name.equals("A"))
                         userID.set("foxtrot");
                      else
                         userID.set("charlie");
                      System.out.println(name + " " + userID.get());
                   }
                };
   Thread thdA = new Thread(r);
   thdA.setName("A");
   Thread thdB = new Thread(r);
   thdB.setName("B");
   thdA.start();
   thdB.start();
}
```

After instantiating ThreadLocal and assigning the reference to a volatile class field named userID (the field is volatile because it is accessed by different threads, which might execute on a multiprocessor/multicore machine), the default main thread creates two more threads that store different `String` objects in `userID` and output their objects.

When you run this application, you will observe the following output (possibly not in this order):

```
A foxtrot
B charlie
```

Values stored in thread-local variables are not related. When a new thread is created, it gets a new storage slot containing `initialValue()`'s value. Perhaps you would prefer to pass a value from a *parent thread*, a thread that creates another thread, to a *child thread*, the created thread. You accomplish this task with `InheritableThreadLocal`.

`InheritableThreadLocal` is a subclass of `ThreadLocal`. In addition to declaring a `public InheritableThreadLocal()` constructor, this class declares the following `protected` method:

- `T childValue(T parentValue)` calculates the child's initial value as a function of the parent's value at the time the child thread is created. This method is called from the parent thread before the child thread is started. The method returns the argument passed to `parentValue` and should be overridden when another value is desired.

Listing 7–16 shows you how to use `InheritableThreadLocal` to pass a parent thread's `Integer` object to a child thread.

Listing 7–16. *Passing an object from parent thread to child thread*

```
private static volatile InheritableThreadLocal<Integer> intVal =
  new InheritableThreadLocal<Integer>();
public static void main(String[] args)
{
   Runnable rP = new Runnable()
                 {
                    @Override
                    public void run()
                    {
                       intVal.set(new Integer(10));
                       Runnable rC = new Runnable()
                                     {
                                        public void run()
                                        {
                                           Thread thd;
                                           thd = Thread.currentThread();
                                           String name = thd.getName();
                                           System.out.println(name + " " +
                                                              intVal.get());
                                        }
                                     };
                       Thread thdChild = new Thread(rC);
                       thdChild.setName("Child");
                       thdChild.start();
                    }
                 };
   new Thread(rP).start();
}
```

After instantiating `InheritableThreadLocal` and assigning it to a `volatile` class field named `intVal`, the default main thread creates a parent thread, which stores an `Integer` object containing 10 in `intVal`. The parent thread creates a child thread, which accesses `intVal` and retrieves its parent thread's `Integer` object.

When you run this application, you will observe the following output:

```
Child 10
```

EXERCISES

The following exercises are designed to test your understanding of this chapter's additional basic APIs:

1. What is reflection?
2. What is the difference between `Class`'s `getDeclaredFields()` and `getFields()` methods?
3. How would you determine if the method represented by a `Method` object is abstract?
4. Identify the three ways of obtaining a `Class` object.
5. True or false: A string literal is a `String` object.
6. What is the purpose of `String`'s `intern()` method?
7. How do `String` and `StringBuffer` differ?
8. How do `StringBuffer` and `StringBuilder` differ?
9. What does `System`'s `arraycopy()` method accomplish?
10. What is a thread?
11. What is the purpose of the `Runnable` interface?
12. What is the purpose of the `Thread` class?
13. True or false: A `Thread` object associates with multiple threads.
14. What is a race condition?
15. What is synchronization?
16. How is synchronization implemented?
17. How does synchronization work?
18. True or false: Variables of type `long` or `double` are not atomic on 32-bit virtual machines.
19. What is the purpose of reserved word `volatile`?
20. True or false: `Object`'s `wait()` methods can be called from outside of a synchronized method or block.
21. What is deadlock?
22. What is the purpose of the `ThreadLocal` class?

23. How does `InheritableThreadLocal` differ from `ThreadLocal`?

24. In Chapter 6, Listing 6-14's demonstration of the `SoftReference` class includes the following array declaration and inefficient loop:

```
String[] imageNames = new String[NUM_IMAGES];
for (int i = 0; i < imageNames.length; i++)
   imageNames[i] = new String("image" + i + ".gif");
```

Rewrite this loop to use `StringBuffer`.

25. `Class` declares `boolean isAnnotation()`, `boolean isEnum()`, and `boolean isInterface()` methods that return true when the `Class` object represents an annotation type, an enum, or an interface, respectively. Create a `Classify` application that uses `Class`'s `forName()` method to load its single command-line argument, which will represent an annotation type, enum, interface, or class (the default). Use a chained if-else statement along with the aforementioned methods to output `Annotation`, `Enum`, `Interface`, or `Class`.

26. The output from Listing 7–1's `ExploreType` application does not look like a class declaration for the `Boolean` class. Improve this application so that `java ExploreType java.lang.Boolean` generates the following output:

```
public class java.lang.Boolean
{
   // FIELDS
   public static final java.lang.Boolean java.lang.Boolean.TRUE
   public static final java.lang.Boolean java.lang.Boolean.FALSE
   public static final java.lang.Class java.lang.Boolean.TYPE
   private final boolean java.lang.Boolean.value
   private static final long java.lang.Boolean.serialVersionUID

   // CONSTRUCTORS
   public java.lang.Boolean(java.lang.String)
   public java.lang.Boolean(boolean)

   // METHODS
   public int java.lang.Boolean.hashCode()
   public boolean java.lang.Boolean.equals(java.lang.Object)
   public int java.lang.Boolean.compareTo(java.lang.Boolean)
   public int java.lang.Boolean.compareTo(java.lang.Object)
   public static boolean java.lang.Boolean.getBoolean(java.lang.String)
   public static java.lang.String java.lang.Boolean.toString(boolean)
   public java.lang.String java.lang.Boolean.toString()
   public static java.lang.Boolean java.lang.Boolean.valueOf(java.lang.String)
   public static java.lang.Boolean java.lang.Boolean.valueOf(boolean)
   public boolean java.lang.Boolean.booleanValue()
   public static boolean java.lang.Boolean.parseBoolean(java.lang.String)
   private static boolean java.lang.Boolean.toBoolean(java.lang.String)
}
```

27. Modify Listing 7–3's `CountingThreads` application by marking the two started threads as daemon threads. What happens when you run the resulting application?

28. Modify Listing 7–3's `CountingThreads` application by adding logic to stop both counting threads when the user presses the Enter key. The default main thread should call `System.in.read()` prior to terminating, and assign `true` to a variable named `stopped` after this method call returns. Each counting thread should test this variable to see if it contains true at the start of each loop iteration, and only continue the loop when the variable contains false.

Summary

The Reflection API lets applications learn about loaded classes, interfaces, enums, and annotation types. The API also lets applications instantiate classes, call methods, access fields, and perform other tasks reflectively.

The entry point into the Reflection API is a special `java.lang` class named `Class`. Additional classes are located in the `java.lang.reflect` package, and include `Constructor`, `Field`, `Method`, `AccessibleObject`, and `Array`.

The `java.lang.String` class represents a string as a sequence of characters. Because instances of this class are immutable, Java provides `java.lang.StringBuffer` for building a string more efficiently.

The `java.lang.System` class provides access to standard input, standard output, and standard error, and other system-oriented resources. For example, `System` provides the `arraycopy()` method as the fastest portable way to copy one array to another.

Finally, Java supports threads via its Threading API. This API consists of one interface (`Runnable`) and four classes (`Thread`, `ThreadGroup`, `ThreadLocal`, and `InheritableThreadLocal`) in the `java.lang` package.

This chapter completes my coverage of Java's basic APIs. Chapter 8 continues to explore Java's foundational APIs by focusing on its utility APIs. Specifically, Chapter 8 introduces you to the collections framework.