



## *How to Use This Quick Reference*

The quick-reference section that follows packs a lot of information into a small space. This introduction explains how to get the most out of that information. It describes how the quick reference is organized and how to read the individual quick-reference entries.

### *Finding a Quick-Reference Entry*

The quick reference is organized into chapters, each of which documents a single package of the Java platform or a group of related packages. Packages are listed alphabetically within and between chapters, so you never really need to know which chapter documents which package: you can simply search alphabetically, as you might do in a dictionary. The documentation for each package begins with a quick-reference entry for the package itself. This entry includes a short overview of the package and a listing of the classes and interfaces included in the package. In this listing of package contents, classes and interfaces are first grouped by general category (interfaces, classes, and exceptions, for example). Within each category, they are grouped by class hierarchy, with indentation to indicate the level of the hierarchy. Finally, classes and interfaces at the same hierarchy level are listed alphabetically.

Each package overview is followed by individual quick-reference entries for the classes and interfaces defined in the package. All the entries in this reference are organized alphabetically by class *and* package name, so related classes are grouped near each other. This means that to look up a quick-reference entry for a particular class, you must also know the name of the package that contains that class. Usually, the package name is obvious from the context, and you should have no trouble looking up the quick-reference entry you want. Use the tabs on the outside edge of the book and the dictionary-style headers on the upper corner of each page to help you quickly find the package and class you need.

Occasionally, you may need to look up a class for which you do not already know the package. In this case, refer to Chapter 25. This index allows you to look up a class by class name and find out what package it is part of.

## *Reading a Quick-Reference Entry*

The quick-reference entries for classes and interfaces contain quite a bit of information. The sections that follow describe the structure of a quick-reference entry, explaining what information is available, where it is found, and what it means. While reading the descriptions that follow, you may find it helpful to flip through the reference section itself to find examples of the features being described.

### *Class Name, Package Name, Availability, and Flags*

Each quick-reference entry begins with a four-part title that specifies the name, package name, and availability of the class, and may also specify various additional flags that describe the class. The class name appears in bold at the upper left of the title. The package name appears, in smaller print, in the lower left, below the class name.

The upper-right portion of the title indicates the availability of the class; it specifies the earliest release that contained the class. If a class was introduced in Java 1.1, for example, this portion of the title reads “Java 1.1”. The availability section of the title is also used to indicate whether a class has been deprecated, and, if so, in what release. For example, it might read “Java 1.1; Deprecated in Java 1.2”.

In the lower-right corner of the title you may find a list of flags that describe the class. The possible flags and their meanings are as follows:

#### *checked*

The class is a checked exception, which means that it extends `java.lang.Exception`, but not `java.lang.RuntimeException`. In other words, it must be declared in the throws clause of any method that may throw it.

#### *cloneable*

The class, or a superclass, implements `java.lang.Cloneable`.

#### *collection*

The class, or a superclass, implements `java.util.Collection` or `java.util.Map`.

#### *comparable*

The class, or a superclass, implements `java.lang.Comparable`.

#### *error*

The class extends `java.lang.Error`.

#### *event*

The class extends `java.util.EventObject`.

#### *event adapter*

The class, or a superclass, implements `java.util.EventListener`, and the class name ends with “Adapter”.

#### *event listener*

The class, or a superclass, implements `java.util.EventListener`.

#### *runnable*

The class, or a superclass, implements `java.lang.Runnable`.

#### *serializable*

The class, or a superclass, implements `java.io.Serializable`.

#### *unchecked*

The class is an unchecked exception, which means it extends `java.lang.RuntimeException` and therefore does not need to be declared in the `throws` clause of a method that may throw it.

### **Description**

The title of each quick-reference entry is followed by a short description of the most important features of the class or interface. This description may be anywhere from a couple of sentences to several paragraphs long.

### **Hierarchy**

If a class or interface has a nontrivial class hierarchy, the “Description” section is followed by a figure that illustrates the hierarchy and helps you understand the class in the context of that hierarchy. The name of each class or interface in the diagram appears in a box; classes appear in rectangles (except for abstract classes, which appear in skewed rectangles or parallelograms). Interfaces appear in rounded rectangles, in which the corners have been replaced by arcs. The current class—the one that is the subject of the diagram—appears in a box that is bolder than the others. The boxes are connected by lines: solid lines indicate an “extends” relationship, and dotted lines indicate an “implements” relationship. The superclass-to-subclass hierarchy reads from left to right in the top row (or only row) of boxes in the figure. Interfaces are usually positioned beneath the classes that implement them, although in simple cases an interface is sometimes positioned on the same line as the class that implements it, resulting in a more compact figure. Note that the hierarchy figure shows only the superclasses of a class. If a class has subclasses, those are listed in the cross-reference section at the end of the quick-reference entry for the class.

### **Synopsis**

The most important part of every quick-reference entry is the class synopsis, which follows the title and description. The synopsis for a class looks a lot like the source code for the class, except that the method bodies are omitted and some additional annotations are added. If you know Java syntax, you know how to read the class synopsis.

The first line of the synopsis contains information about the class itself. It begins with a list of class modifiers, such as `public`, `abstract`, and `final`. These modifiers are followed by the class or interface keyword and then by the name of the class. The class name may be followed by an `extends` clause that specifies the superclass and an `implements` clause that specifies any interfaces the class implements.

The class definition line is followed by a list of the fields and methods that the class defines. Once again, if you understand basic Java syntax, you should have no trouble making sense of these lines. The listing for each member includes the modifiers, type, and name of the member. For methods, the synopsis also includes the type and name of each method parameter and an optional throws clause that lists the exceptions the method can throw. The member names are in boldface, so it is easy to scan the list of members looking for the one you want. The names of method parameters are in italics to indicate that they are not to be used literally. The member listings are printed on alternating gray and white backgrounds to keep them visually separate.

### *Member availability and flags*

Each member listing is a single line that defines the API for that member. These listings use Java syntax, so their meaning is immediately clear to any Java programmer. There is some auxiliary information associated with each member synopsis, however, that requires explanation.

Recall that each quick-reference entry begins with a title section that includes the release in which the class was first defined. When a member is introduced into a class after the initial release of the class, the version in which the member was introduced appears, in small print, to the left of the member synopsis. For example, if a class was first introduced in Java 1.1, but had a new method added in Java 1.2 the title contains the string “Java 1.1”, and the listing for the new member is preceded by the number “1.2”. Furthermore, if a member has been deprecated, that fact is indicated with a hash mark (#) to the left of the member synopsis.

The area to the right of the member synopsis is used to display a variety of flags that provide additional information about the member. Some of these flags indicate additional specification details that do not appear in the member API itself. Other flags contain implementation-specific information. This information can be quite useful in understanding the class and in debugging your code, but be aware that it may differ between implementations. The implementation-specific flags displayed in this book are based on Sun’s Linux implementation of Java.

The following flags may be displayed to the right of a member synopsis:

#### *native*

An implementation-specific flag that indicates that a method is implemented in native code. Although `native` is a Java keyword and can appear in method signatures, it is part of the method implementation, not part of its specification. Therefore, this information is included with the member flags, rather than as part of the member listing. This flag is useful as a hint about the expected performance of a method.

#### *synchronized*

An implementation-specific flag that indicates that a method implementation is declared `synchronized`, meaning that it obtains a lock on the object or class before executing. Like the `native` keyword, the `synchronized` keyword is part of the method implementation, not part of the specification, so it appears as a flag, not in the method synopsis itself. This flag is a useful hint that the method is probably implemented in a thread-safe manner.

Whether or not a method is thread-safe is part of the method specification, and this information *should* appear (although it often does not) in the method documentation. There are a number of different ways to make a method thread-safe, however, and declaring the method with the `synchronized` keyword is only one possible implementation. In other words, a method that does not bear the `synchronized` flag can still be thread-safe.

*Overrides:*

This flag indicates that a method overrides a method in one of its super-classes. The flag is followed by the name of the superclass that the method overrides. This is a specification detail, not an implementation detail. As we'll see in the next section, overriding methods are usually grouped together in their own section of the class synopsis. The `Overrides:` flag is only used when an overriding method is not grouped in that way.

*Implements:*

This flag indicates that a method implements a method in an interface. The flag is followed by the name of the interface that is implemented. This is a specification detail, not an implementation detail. As we'll see in the next section, methods that implement an interface are usually grouped into a special section of the class synopsis. The `Implements:` flag is only used for methods that are not grouped in this way.

*empty*

This flag indicates that the implementation of the method has an empty body. This can be a hint to the programmer that the method may need to be overridden in a subclass.

*constant*

An implementation-specific flag that indicates that a method has a trivial implementation. Only methods with a `void` return type can be truly empty. Any method declared to return a value must have at least a return statement. The “constant” flag indicates that the method implementation is empty except for a return statement that returns a constant value. Such a method might have a body like `return null;` or `return false;`. Like the “empty” flag, this flag may indicate that a method needs to be overridden.

*default:*

This flag is used with property accessor methods that read the value of a property (i.e., methods whose names begin with “get” and take no arguments). The flag is followed by the default value of the property. Strictly speaking, default property values are a specification detail. In practice, however, these defaults are not always documented, and care should be taken, because the default values may change between implementations.

Not all property accessors have a “default:” flag. A default value is determined by dynamically loading the class in question, instantiating it using a no-argument constructor, and then calling the method to find out what it returns. This technique can be used only on classes that can be dynamically loaded and instantiated and that have no-argument constructors, so default values are shown for those classes only. Furthermore, note that when a class is instanti-

ated using a different constructor, the default values for its properties may be different.

=

For `static final` fields, this flag is followed by the constant value of the field. Only constants of primitive and `String` types and constants with the value `null` are displayed. Some constant values are specification details, while others are implementation details. The reason that symbolic constants are defined, however, is so you can write code that does not rely directly upon the constant value. Use this flag to help you understand the class, but do not rely upon the constant values in your own programs.

### *Functional grouping of members*

Within a class synopsis, the members are not listed in strict alphabetical order. Instead, they are broken down into functional groups and listed alphabetically within each group. Constructors, methods, fields, and inner classes are all listed separately. Instance methods are kept separate from static (class) methods. Constants are separated from non-constant fields. Public members are listed separately from protected members. Grouping members by category breaks a class down into smaller, more comprehensible segments, making the class easier to understand. This grouping also makes it easier for you to find a desired member.

Functional groups are separated from each other in a class synopsis with Java comments, such as “// Public Constructors”, “// Inner Classes”, and “// Methods Implementing `DataInput`”. The various functional categories are as follows (in the order in which they appear in a class synopsis):

#### *Constructors*

Displays the constructors for the class. Public constructors and protected constructors are displayed separately in subgroupings. If a class defines no constructor at all, the Java compiler adds a default no-argument constructor that is displayed here. If a class defines only private constructors, it cannot be instantiated, so a special, empty grouping entitled “No Constructor” indicates this fact. Constructors are listed first because the first thing you do with most classes is instantiate them by calling a constructor.

#### *Constants*

Displays all of the constants (i.e., fields that are declared `static` and `final`) defined by the class. Public and protected constants are displayed in separate subgroupings. Constants are listed here, near the top of the class synopsis, because constant values are often used throughout the class as legal values for method parameters and return values.

#### *Inner classes*

Groups all of the inner classes and interfaces defined by the class or interface. For each inner class, there is a single-line synopsis. Each inner class also has its own quick-reference entry that includes a full class synopsis for the inner class. Like constants, inner classes are listed near the top of the class synopsis because they are often used by a number of other members of the class.

#### *Static methods*

Lists the static methods (class methods) of the class, broken down into subgroups for public static methods and protected static methods.

#### *Event listener registration methods*

Lists the public instance methods that register and deregister event listener objects with the class. The names of these methods begin with the words “add” and “remove” and end in “Listener”. These methods are always passed a `java.util.EventListener` object. The methods are typically defined in pairs, so the pairs are listed together. The methods are listed alphabetically by event name rather than by method name.

#### *Property accessor methods*

Lists the public instance methods that set or query the value of a property or attribute of the class. The names of these methods begin with the words “set”, “get”, and “is”, and their signatures follow the patterns set out in the JavaBeans specification. Although the naming conventions and method signature patterns are defined for JavaBeans, classes and interfaces throughout the Java platform define property accessor methods that follow these conventions and patterns. Looking at a class in terms of the properties it defines can be a powerful tool for understanding the class, so property methods are grouped together in this section. Property accessor methods are listed alphabetically by property name, not by method name. This means that the “set”, “get”, and “is” methods for a property all appear together.

#### *Public instance methods*

Contains all of the public instance methods that are not grouped elsewhere.

#### *Implementing methods*

Groups the methods that implement the same interface. There is one subgroup for each interface implemented by the class. Methods that are defined by the same interface are almost always related to each other, so this is a useful functional grouping of methods.

Note that if an interface method is also an event registration method or a property accessor method, it is listed both in this group and in the event or property group. This situation does not arise often, but when it does, all of the functional groupings are important and useful enough to warrant the duplicate listing. When an interface method is listed in the event or property group, it displays an “Implements:” flag that specifies the name of the interface of which it is part.

#### *Overriding methods*

Groups the methods that override methods of a superclass broken down into subgroups by superclass. This is typically a useful grouping, because it helps to make it clear how a class modifies the default behavior of its superclasses. In practice, it is also often true that methods that override the same superclass are functionally related to each other.

Sometimes a method that overrides a superclass is also a property accessor method or (more rarely) an event registration method. When this happens, the method is grouped with the property or event methods and displays a flag that indicates which superclass it overrides. The method is not listed with

other overriding methods, however. Note that this is different from interface methods, which, because they are more strongly functionally related, may have duplicate listings in both groups.

*Protected instance methods*

Contains all of the protected instance methods that are not grouped elsewhere.

*Fields*

Lists all the non-constant fields of the class, breaking them down into subgroups for public and protected static fields and public and protected instance fields. Many classes do not define any publicly accessible fields. For those that do, many object-oriented programmers prefer not to use those fields directly, but instead to use accessor methods when such methods are available.

*Deprecated members*

Deprecated methods and deprecated fields are grouped at the very bottom of the class synopsis. Use of these members is strongly discouraged.

## ***Cross-References***

The synopsis section of a quick-reference entry is followed by a number of optional cross-reference sections that indicate other, related classes and methods that may be of interest. These sections are the following:

*Subclasses*

This section lists the subclasses of this class, if there are any.

*Implementations*

This section lists classes that implement this interface.

*Passed To*

This section lists all of the methods and constructors that are passed an object of this type as an argument. This is useful when you have an object of a given type and want to figure out what you can do with it.

*Returned By*

This section lists all of the methods (but not constructors) that return an object of this type. This is useful when you know that you want to work with an object of this type, but don't know how to obtain one.

*Thrown By*

For checked exception classes, this section lists all of the methods and constructors that throw exceptions of this type. This material helps you figure out when a given exception or error may be thrown. Note, however, that this section is based on the exception types listed in the throws clauses of methods and constructors. Subclasses of `RuntimeException` and `Error` do not have to be listed in throws clauses, so it is not possible to generate a complete cross reference of methods that throw these types of unchecked exceptions.

*Type Of*

This section lists all of the fields and constants that are of this type, which can help you figure out how to obtain an object of this type.



### *A Note About Class Names*

Throughout the quick reference, you'll notice that classes are sometimes referred to by class name alone and at other times referred to by class name and package name. If package names were always used, the class synopses would become long and hard to read. On the other hand, if package names were never used, it would sometimes be difficult to know what class was being referred to. The rules for including or omitting the package name are complex. They can be summarized approximately as follows, however:

- If the class name alone is ambiguous, the package name is always used.
- If the class is part of the `java.lang` package or is a very commonly used class, such as `java.io.Serializable`, the package name is omitted.
- If the class being referred to is part of the current package (and has a quick-reference entry in the current chapter), the package name is omitted.