Table of Contents

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Chapter 4. Java 5.0 Language Features

This chapter covers the three most important new language features of Java 5.0. *Generics* add type-safety and expressiveness to Java programs by allowing types to be parameterized with other types. A `List` that contains `String` objects, for example, can be written as `List<String>`. Using parameterized types makes Java code clearer and allows us to remove most casts from our programs.

Enumerated types, or *enums*, are a new category of reference type, like classes and interfaces. An enumerated type defines a finite ("enumerated") set of values, and, importantly, provides type-safety: a variable of enumerated type can hold only values of that enumerated type or `null`. Here is a simple enumerated type definition:

```
public enum Seasons { WINTER, SPRING, SUMMER, AUTUMN }
```

The third Java 5.0 feature discussed in this chapter is program annotations and the annotation types that define them. An *annotation* associates arbitrary data (or metadata) with a program element such as a class, method, field, or even a method parameter or local variable. The type of data held in an annotation is defined by its *annotation type*, which, like enumerated types, is another new category of reference type. The Java 5.0 platform includes three standard annotation types used to provide additional information to the Java compiler. Annotations will probably find their greatest use with code generation tools in Java enterprise programming.

Java 5.0 also introduces a number of other important new language features that don't require a special chapter to explain. Coverage of these changes is found in sections throughout Chapter 2. They include:

- Autoboxing and unboxing conversions
- The `for/in` looping statement, sometimes called "foreach"
- Methods with variable-length argument lists, also known as *varargs* methods
- The ability to narrow the return type of a method when overriding, known as a " covariant return"
- The `import static` directive, which imports the static members of a type into the namespace

4.1. Generic Types

Generic types and methods are the defining new feature of Java 5.0. A *generic type* is defined using one or more *type variables* and has one or more methods that use a type

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

variable as a placeholder for an argument or return type. For example, the type `java.util.List<E>` is a generic type: a list that holds elements of some type represented by the placeholder `E`. This type has a method named `add()`, declared to take an argument of type `E`, and a method named `get()`, declared to return a value of type `E`.

In order to use a generic type like this, you specify actual types for the type variable (or variables), producing a *parameterized type* such as `List<String>`.[1] The reason to specify this extra type information is that the compiler can provide much stronger compile-time type checking for you, increasing the type safety of your programs. This type checking prevents you from adding a `String[]`, for example, to a `List` that is intended to hold only `String` objects. Also, the additional type information enables the compiler to do some casting for you. The compiler knows that the `get( )` method of a `List<String>` (for example) returns a `String` object: you are no longer required to cast a return value of type `Object` to a `String`.

[1] Throughout this chapter, I've tried to consistently use the term " generic type" to mean a type that declares one or more type variables and the term "parameterized type" to mean a generic type that has had actual type arguments substituted for its type varaiables. In common usage, however, the distinction is not a sharp one and the terms are sometimes used interchangeably.

The collections classes of the `java.util` package have been made generic in Java 5.0, and you will probably use them frequently in your programs. Typesafe collections are the canonical use case for generic types. Even if you never define generic types of your own and never use generic types other than the collections classes in `java.util`, the benefits of typesafe collections are so significant that they justify the complexity of this major new language feature.

We begin by exploring the basic use of generics in typesafe collections, then delve into more complex details about the use of generic types. Next we cover type parameter wildcards and bounded wildcards. After describing how to use generic types, we explain how to write your own generic types and generic methods. Our coverage of generics concludes with a tour of important generic types in the core Java API. It explores these types and their use in depth in order to provide a deeper understanding of how generics work.

4.1.1. Typesafe Collections

The `java.util` package includes the Java Collections Framework for working with sets and lists of objects and mappings from key objects to value objects. Collections are covered in Chapter 5. Here, we discuss the fact that in Java 5.0 the collections classes use type parameters to identify the type of the objects in the collection. This is not the case in Java 1.4 and earlier. Without generics, the use of collections requires the programmer to remember the proper element type for each collection. When you create a collection in Java 1.4, you know what type of objects you intend to store in that collection, but the compiler cannot know this. You must be careful to add elements of the appropriate type.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

And when querying elements from a collection, you must write explicit casts to convert them from `Object` to their actual type. Consider the following Java 1.4 code:

```
public static void main(String[] args) {
    // This list is intended to hold only strings.
    // The compiler doesn't know that so we have to remember ourselves.
    List wordlist = new ArrayList();

    // Oops! We added a String[] instead of a String.
    // The compiler doesn't know that this is an error.
    wordlist.add(args);

    // Since List can hold arbitrary objects, the get() method returns
    // Object.  Since the list is intended to hold strings, we cast the
    // return value to String but get a ClassCastException because of
    // the error above.
    String word = (String)wordlist.get(0);
}
```

Generic types solve the type safety problem illustrated by this code. `List` and the other collection classes in `java.util` have been rewritten to be generic. As mentioned above, `List` has been redefined in terms of a type variable named `E` that represents the type of the elements of the list. The `add( )` method is redefined to expect an argument of type `E` instead of `Object` and `get( )` has been redefined to return `E` instead of `Object`.

In Java 5.0, when we declare a `List` variable or create an instance of an `ArrayList`, we specify the actual type we want `E` to represent by placing the actual type in angle brackets following the name of the generic type. A `List` that holds strings is a `List<String>`, for example. Note that this is much like passing an argument to a method, except that we use types rather than values and angle brackets instead of parentheses.

The elements of the `java.util` collection classes must be objects; they cannot be used with primitive values. The introduction of generics does not change this. Generics do not work with primitives: we can't declare a `Set<char>`, or a `List<int>` for example. Note, however, that the autoboxing and autounboxing features of Java 5.0 make working with a `Set<Character>` or a `List<Integer>` just as easy as working directly with `char` and `int` values. (See Chapter 2 for details on autoboxing and autounboxing).

In Java 5.0, the example above would be rewritten as follows:

```
public static void main(String[] args) {
    // This list can only hold String objects
    List<String> wordlist = new ArrayList<String>();

    // args is a String[], not String, so the compiler won't let us do this
    wordlist.add(args);  // Compilation error!

    // We can do this, though.
    // Notice the use of the new for/in looping statement
    for(String arg : args) wordlist.add(arg);

    // No cast is required.  List<String>.get() returns a String.
    String word = wordlist.get(0);
}
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Note that this code isn't much shorter than the nongeneric example it replaces. The cast, which uses the word `String` in parentheses, is replaced with the type parameter, which places the word `String` in angle brackets. The difference is that the type parameter has to be declared only once, but the list can be used any number of times without a cast. This would be more apparent in a longer example. But even in cases where the generic syntax is more verbose than the nongeneric syntax, it is still very much worth using generics because the extra type information allows the compiler to perform much stronger error checking on your code. Errors that would only be apparent at runtime can now be detected at compile time. Furthermore, the compilation error appears at the exact line where the type safety violation occurs. Without generics, a `ClassCastException` can be thrown far from the actual source of the error.

Just as methods can have any number of arguments, classes can have more than one type variable. The `java.util.Map` interface is an example. A `Map` is a mapping from key objects to value objects. The `Map` interface declares one type variable to represent the type of the keys and one variable to represent the type of the values. As an example, suppose you want to map from `String` objects to `Integer` objects:

```
public static void main(String[] args) {
    // A map from strings to their position in the args[] array
    Map<String,Integer> map = new HashMap<String,Integer>();

    // Note that we use autoboxing to wrap i in an Integer object.
    for(int i=0; i < args.length; i++) map.put(args[i], i);

    // Find the array index of a word.  Note no cast is required!
    Integer position = map.get("hello");

    // We can also rely on autounboxing to convert directly to an int,
    // but this throws a NullPointerException if the key does not exist
    // in the map
    int pos = map.get("world");
}
```

A parameterized type like `List<String>` is itself a type and can be used as the value of a type parameter for some other type. You might see code like this:

```
// Look at all those nested angle brackets!
Map<String, List<List<int[]>>> map = getWeirdMap();

// The compiler knows all the types and we can write expressions
// like this without casting.  We might still get NullPointerException
// or ArrayIndexOutOfBounds at runtime, of course.
int value = map.get(key).get(0).get(0)[0];

// Here's how we break that expression down step by step.
List<List<int[]>> listOfLists = map.get(key);
List<int[]> listOfIntArrays = listOfLists.get(0);
int[] array = listOfIntArrays.get(0);
int element = array[0];
```

In the code above, the `get( )` methods of `java.util.List<E>` and `java.util.Map<K,V>` return a list or map element of type `E` and `V` respectively. Note, however, that generic types can use their variables in more sophisticated ways. Look up

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

`List<E>` in the reference section of this book, and you'll find that its `iterator( )` method is declared to return an `Iterator<E>`. That is, the method returns an instance of a parameterized type whose actual type parameter is the same as the actual type parameter of the list. To illustrate this concretely, here is a way to obtain the first element of a `List<String>` without calling `get(0)`.

```
List<String> words = // ...initialized elsewhere...
Iterator<String> iterator = words.iterator();
String firstword = iterator.next();
```

4.1.2. Understanding Generic Types

This section delves deeper into the details of generic type usage, explaining the following topics:

- The consequences of using generic types *without* type parameters
- The parameterized type hierarchy
- A hole in the compile-time type safety of generic types and a patch to ensure runtime type safety
- Why arrays of parameterized types are not typesafe

4.1.2.1. Raw types and unchecked warnings

Even though the Java collection classes have been modified to take advantage of generics, you are not required to specify type parameters to use them. A generic type used without type parameters is known as a *raw type*. Existing pre-5.0 code continues to work: you simply write all the casts that you're already used to writing, and you put up with some pestering from the compiler. Consider the following code that stores objects of mixed types into a raw `List`:

```
List l = new ArrayList();
l.add("hello");
l.add(new Integer(123));
Object o = l.get(0);
```

This code works fine in Java 1.4. If we compile it using Java 5.0, however, *javac* compiles the code but prints this complaint:

```
Note: Test.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

When we recompile with the `-Xlint` option as suggested, we see these warnings:

```
Test.java:6: warning: [unchecked]
    unchecked call to add(E) as a member of the raw type java.util.List
        l.add("hello");
         ^
Test.java:7: warning: [unchecked]
    unchecked call to add(E) as a member of the raw type java.util.List
        l.add(new Integer(123));
         ^
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

The compiler warns us about the `add( )` calls because it cannot ensure that the values being added to the list have the correct types. It is letting us know that because we've used a raw type, it cannot verify that our code is typesafe. Note that the call to `get( )` is okay because it is extracting an element that is already safely in the list.

If you get unchecked warnings on files that do not use any of the new Java 5.0 features, you can simply compile them with the `-source 1.4` flag, and the compiler won't complain. If you can't do that, you can ignore the warnings, suppress them with an `@SuppressWarnings("unchecked")` annotation (see Section 4.3 later in this chapter) or upgrade your code to specify a type parameter.[2] The following code, for example, compiles with no warnings and still allows you to add objects of mixed types to the list:

[2] At the time of this writing, *javac* does not yet honor the `@SuppressWarnings` annotation. It is expected to do so in Java 5.1.

```
List<Object> l = new ArrayList<Object>();
l.add("hello");
l.add(123);             // autoboxing
Object o = l.get(0);
```

4.1.2.2. The parameterized type hierarchy

Parameterized types form a type hierarchy, just as normal types do. The hierarchy is based on the base type, however, and not on the type of the parameters. Here are some experiments you can try:

```
ArrayList<Integer> l = new ArrayList<Integer>();
List<Integer> m = l;                          // okay
Collection<Integer> n = l;                    // okay
ArrayList<Number> o = l;                      // error
Collection<Object> p = (Collection<Object>)l;   // error, even with cast
```

A `List<Integer>` is a `Collection<Integer>`, but it is not a `List<Object>`. This is nonintuitive, and it is important to understand why generics work this way. Consider this code:

```
List<Integer> li = new ArrayList<Integer>();
li.add(123);

// The line below will not compile.  But for the purposes of this
// thought-experiment, assume that it does compile and see how much
// trouble we get ourselves into.
List<Object> lo = li;

// Now we can retrieve elements of the list as Object instead of Integer
Object number = lo.get(0);

// But what about this?
lo.add("hello world");

// If the line above is allowed then the line below throws ClassCastException
Integer i = li.get(1);  // Can't cast a String to Integer!
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

This then is the reason that a `List<Integer>` is not a `List<Object>`, even though all elements of a `List<Integer>` are in fact instances of `Object`. If the conversion to `List<Object>` were allowed, non-`Integer` objects could be added to the list.

4.1.2.3. Runtime type safety

As we've seen, a `List<X>` cannot be converted to a `List<Y>`, even when `X` *can* be converted to `Y`. A `List<X>` can be converted to a `List`, however, so that you can pass it to a legacy method that expects an argument of that type and has not been updated for generics.

This ability to convert parameterized types to nonparameterized types is essential for backward compatibility, but it does open up a hole in the type safety system that generics offer:

```
// Here's a basic parameterized list.
List<Integer> li = new ArrayList<Integer>();

// It is legal to assign a parameterized type to a nonparameterized variable
List l = li;

// This line is a bug, but it compiles and runs.
// The Java 5.0 compiler will issue an unchecked warning about it.
// If it appeared as part of a legacy class compiled with Java 1.4, however,
// then we'd never even get the warning.
l.add("hello");

// This line compiles without warning but throws ClassCastException at runtime.
// Note that the failure can occur far away from the actual bug.
Integer i = li.get(0);
```

Generics provide compile-time type safety only. If you compile all your code with the Java 5.0 compiler and do not get any unchecked warnings, these compile-time checks are enough to ensure that your code is also typesafe at runtime. But if you have unchecked warnings or are working with legacy code that manipulates your collections as raw types, you may want to take additional steps to ensure type safety at runtime. You can do this with methods like `checkedList()` and `checkedMap( )` of `java.util.Collections`. These methods enclose your collection in a wrapper collection that performs runtime type checks to ensure that only values of the correct type are added to the collection. For example, we could prevent the type safety hole shown above like this:

```
// Here's a basic parameterized list.
List<Integer> li = new ArrayList<Integer>();

// Wrap it for runtime type safety
List<Integer> cli = Collections.checkedList(li, Integer.class);

// Now widen the checked list to the raw type
List l = cli;

// This line compiles but fails at runtime with a ClassCastException.
// The exception occurs exactly where the bug is, rather than far away
l.add("hello");
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

4.1.2.4. Arrays of parameterized type

Arrays require special consideration when working with generic types. Recall that an array of type `S[ ]` is also of type `T[]`, if `T` is a superclass (or interface) of `S`. Because of this, the Java interpreter must perform a runtime check every time you store an object in an array to ensure that the runtime type of the object and of the array are compatible. For example, the following code fails this runtime check and throws an `ArrayStoreException`:

```
String[] words = new String[10];
Object[] objs = words;
objs[0] = 1;  // 1 autoboxed to an Integer, throws ArrayStoreException
```

Although the compile-time type of `objs` is `Object[]`, its runtime type is `String[ ]`, and it is not legal to store an `Integer` in it.

When we work with generic types, the runtime check for array store exceptions is no longer sufficient because a check performed at runtime does not have access to the compile-time type parameter information. Consider this (hypothetical) code:

```
List<String>[] wordlists = new ArrayList<String>[10];
ArrayList<Integer> ali = new ArrayList<Integer>();
ali.add(123);
Object[] objs = wordlists;
objs[0] = ali;                        // No ArrayStoreException
String s = wordlists[0].get(0);       // ClassCastException!
```

If the code above were allowed, the runtime array store check would succeed: without compile-time type parameters, the code simply stores an `ArrayList` into an `ArrayList[]` array, which is perfectly legal. Since the compiler can't prevent you from defeating type safety in this way, it instead prevents you from creating any array of parameterized type. The scenario above can never occur because the compiler will refuse to compile the first line.

Note that this is not a blanket restriction on using arrays with generics; it is just a restriction on creating arrays of parameterized type. We'll return to this issue when we look at how to write generic methods.

4.1.3. Type Parameter Wildcards

Suppose we want to write a method to display the elements of a `List`.[3] Before `List` was a generic type, we'd just write code like this:

[3] The three `printList()` methods shown in this section ignore the fact that the `List` implementations classes in `java.util` all provide working `toString()` methods. Notice also that the methods assume that the `List` implements `RandomAccess` and provides very poor performance on `LinkedList` instances.

```
public static void printList(PrintWriter out, List list) {
    for(int i=0, n=list.size(); i < n; i++) {
        if (i > 0) out.print(", ");
        out.print(list.get(i).toString());
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    }
}
```

In Java 5.0, `List` is a generic type, and, if we try to compile this method, we'll get unchecked warnings. In order to get rid of those warnings, you might be tempted to modify the method as follows:

```
public static void printList(PrintWriter out, List<Object> list) {
    for(int i=0, n=list.size(); i < n; i++) {
        if (i > 0) out.print(", ");
        out.print(list.get(i).toString());
    }
}
```

This code compiles without warnings but isn't very useful because the only lists that can be passed to it are lists explicitly declared of type `List<Object>`. Remember that `List<String>` and `List<Integer>` (for example) cannot be widened or cast to `List<Object>`. What we really want is a typesafe `printList()` method to which we can pass any `List`, regardless of how it has been parameterized. The solution is to use a wildcard as the type parameter. The method would then be written like this:

```
public static void printList(PrintWriter out, List<?> list) {
    for(int i=0, n=list.size(); i < n; i++) {
        if (i > 0) out.print(", ");
        Object o = list.get(i);
        out.print(o.toString());
    }
}
```

This version of the method compiles without warnings and can be used the way we want it to be used. The `?` wildcard represents an unknown type, and the type `List<?>` is read as "List of unknown."

As a general rule, if a type is generic and you don't know or don't care about the value of the type variable, you should always use a `?` wildcard instead of using a raw type. Raw types are allowed only for backward compatibility and should be used only in legacy code. Note, however, that you cannot use a wildcard when invoking a constructor. The following code is not legal:

```
List<?> l = new ArrayList<?>();
```

There is no sense in creating a `List` of unknown type. If you are creating it, you should know what kind of elements it will hold. You may later want to pass such a list to a method that does not care about its element type, but you need to specify an element type when you create it. If what you really want is a `List` that can hold any type of object, do this:

```
List<Object> l = new ArrayList<Object>();
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

It should be clear from the `printList( )` variants above that a `List<?>` is not the same thing as a `List<Object>` and that neither is the same thing as a raw `List`. A `List<?>` has two important properties that result from the use of a wildcard. First, consider methods like `get()` that are declared to return a value of the same type as the type parameter. In this case, that type is unknown, so these methods return an `Object`. Since all we need to do with the object is invoke its `toString()` method, this is fine for our needs.

Second, consider `List` methods such as `add()` that are declared to accept an argument whose type is specified by the type parameter. This is the more surprising case: when the type parameter is unknown, the compiler does not let you invoke any methods that have a parameter of the unknown type because it cannot check that you are passing an appropriate value. A `List<?>` is effectively read-only since the compiler does not allow us to invoke methods like `add( )`, `set()`, and `addAll( )`.

4.1.3.1. Bounded wildcards

Let's continue now with a slightly more complex variant of our original example. Suppose that we want to write a `sumList()` method to compute the sum of a list of `Number` objects. As before, we could use a raw `List`, but we would give up type safety and have to deal with unchecked warnings from the compiler. Or we could use a `List<Number>`, but then we wouldn't be able to call the method for a `List<Integer>` or `List<Double>`, types we are more likely to use in practice. But if we use a wildcard, we don't actually get the type safety that we want because we have to trust that our method will be called with a `List` whose type parameter is actually `Number` or a subclass and not, say, a `String`. Here's what such a method might look like:

```
public static double sumList(List<?> list) {
    double total = 0.0;
    for(Object o : list) {
        Number n = (Number) o;  // A cast is required and may fail
        total += n.doubleValue();
    }
    return total;
}
```

To fix this method and make it truly typesafe, we need to use a *bounded wildcard* that states that the type parameter of the `List` is an unknown type that is either `Number` or a subclass of `Number`. The following code does just what we want:

```
public static double sumList(List<? extends Number> list) {
    double total = 0.0;
    for(Number n : list) total += n.doubleValue();
    return total;
}
```

The type `List<? extends Number>` could be read as "`List` of unknown descendant of `Number`." It is important to understand that, in this context, `Number` is considered a descendant of itself.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Note that the cast is no longer required. We don't know the type of the elements of the list, but we know that they have an "upper bound" of `Number` so we can extract them from the list as `Number` objects. The use of a `for/in` loop obscures the process of extracting elements from a list somewhat. The general rule is that when you use a bounded wildcard with an upper bound, methods (like the `get()` method of `List`) that return a value of the type parameter use the upper bound. So if we called `list.get( )` instead of using a `for/in` loop, we'd also get a `Number`. The prohibition on calling methods like `list.add ( )` that have arguments of the type parameter type still stands: if the compiler allowed us to call those methods we could add an `Integer` to a list that was declared to hold only `Short` values, for example.

It is also possible to specify a lower-bounded wildcard using the keyword `super` instead of `extends`. This technique has a different impact on what methods can be called. Lower-bounded wildcards are much less commonly used than upper-bounded wildcards, and we discuss them later in the chapter.

4.1.4. Writing Generic Types and Methods

Creating a simple generic type is straightforward. First, declare your type variables by enclosing a comma-separated list of their names within angle brackets after the name of the class or interface. You can use those type variables anywhere a type is required in any instance fields or methods of the class. Remember, though, that type variables exist only at compile time, so you can't use a type variable with the runtime operators `instanceof` and `new`.

We begin this section with a simple generic type, which we will subsequently refine. This code defines a `Tree` data structure that uses the type variable `V` to represent the type of the value held in each node of the tree:

```
import java.util.*;

/**
 * A tree is a data structure that holds values of type V.
 * Each tree has a single value of type V and can have any number of
 * branches, each of which is itself a Tree.
 */
public class Tree<V> {
    // The value of the tree is of type V.
    V value;

    // A Tree<V> can have branches, each of which is also a Tree<V>
    List<Tree<V>> branches = new ArrayList<Tree<V>>();

    // Here's the constructor.  Note the use of the type variable V.
    public Tree(V value) { this.value = value; }

    // These are instance methods for manipulating the node value and branches.
    // Note the use of the type variable V in the arguments or return types.
    V getValue() { return value; }
    void setValue(V value) { this.value = value; }
    int getNumBranches() { return branches.size(); }
    Tree<V> getBranch(int n) { return branches.get(n); }
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    void addBranch(Tree<V> branch) { branches.add(branch); }
}
```

As you've probably noticed, the naming convention for type variables is to use a single capital letter. The use of a single letter distinguishes these variables from the names of actual types since real-world types always have longer, more descriptive names. The use of a capital letter is consistent with type naming conventions and distinguishes type variables from local variables, method parameters, and fields, which are sometimes written with a single lowercase letter. Collection classes like those in `java.util` often use the type variable `E` for "Element type." When a type variable can represent absolutely anything, `T` (for Type) and `S` are used as the most generic type variable names possible (like using `i` and `j` as loop variables).

Notice that the type variables declared by a generic type can be used only by the instance fields and methods (and nested types) of the type and not by static fields and methods. The reason, of course, is that it is instances of generic types that are parameterized. Static members are shared by all instances and parameterizations of the class, so static members do not have type parameters associated with them. Methods, including static methods, can declare and use their own type parameters, however, and each invocation of such a method can be parameterized differently. We'll cover this later in the chapter.

4.1.4.1. Type variable bounds

The type variable `V` in the declaration above of the `Tree<V>` class is unconstrained: `Tree` can be parameterized with absolutely any type. Often we want to place some constraints on the type that can be used: we might want to enforce that a type parameter implements one or more interfaces, or that it is a subclass of a specified class. This can be done by specifying a *bound* for the type variable. We've already seen upper bounds for wildcards, and upper bounds can also be specified for type variables using a similar syntax. The following code is the `Tree` example rewritten to make `Tree` objects `Serializable` and `Comparable`. In order to do this, the example uses a type variable bound to ensure that its value type is also `Serializable` and `Comparable`. Note how the addition of the `Comparable` bound on `V` enables us to write the `compareTo()` method `Tree` by guaranteeing the existence of a `compareTo()` method on `V`.[4]

[4] The bound shown here requires that the value type `V` is comparable to itself, in other words, that it implements the `Comparable` interface directly. This rules out the use of types that inherit the `Comparable` interface from a superclass. We'll consider the `Comparable` interface in much more detail at the end of this section and present an alternative there.

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

public class Tree<V extends Serializable & Comparable<V>>
    implements Serializable, Comparable<Tree<V>>
{
    V value;
    List<Tree<V>> branches = new ArrayList<Tree<V>>();

    public Tree(V value) { this.value = value; }
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    // Instance methods
    V getValue() { return value; }
    void setValue(V value) { this.value = value; }
    int getNumBranches() { return branches.size(); }
    Tree<V> getBranch(int n) { return branches.get(n); }
    void addBranch(Tree<V> branch) { branches.add(branch); }

    // This method is a nonrecursive implementation of Comparable<Tree<V>>
    // It only compares the value of this node and ignores branches.
    public int compareTo(Tree<V> that) {
        if (this.value == null && that.value == null) return 0;
        if (this.value == null) return -1;
        if (that.value == null) return 1;
        return this.value.compareTo(that.value);
    }

    // javac -Xlint warns us if we omit this field in a Serializable class
    private static final long serialVersionUID = 833546143621133467L;
}
```

The bounds of a type variable are expressed by following the name of the variable with the word `extends` and a list of types (which may themselves be parameterized, as `Comparable` is). Note that with more than one bound, as in this case, the bound types are separated with an ampersand rather than a comma. Commas are used to separate type variables and would be ambiguous if used to separate type variable bounds as well. A type variable can have any number of bounds, including any number of interfaces and at most one class.

4.1.4.2. Wildcards in generic types

Earlier in the chapter we saw examples using wildcards and bounded wildcards in methods that manipulated parameterized types. They are also useful in generic types. Our current design of the `Tree` class requires the value object of every node to have exactly the same type, `V`. Perhaps this is too strict, and we should allow branches of a tree to have values that are a subtype of `V` instead of requiring `V` itself. This version of the `Tree` class (minus the `Comparable` and `Serializable` implementation) is more flexible:

```
public class Tree<V> {
    // These fields hold the value and the branches
    V value;
    List<Tree<? extends V>> branches = new ArrayList<Tree<? extends V>>();

    // Here's a constructor
    public Tree(V value) { this.value = value; }

    // These are instance methods for manipulating value and branches
    V getValue() { return value; }
    void setValue(V value) { this.value = value; }
    int getNumBranches() { return branches.size(); }
    Tree<? extends V> getBranch(int n) { return branches.get(n); }
    void addBranch(Tree<? extends V> branch) { branches.add(branch); }
}
```

The use of bounded wildcards for the branch type allow us to add a `Tree<Integer>`, for example, as a branch of a `Tree<Number>`:

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
Tree<Number> t = new Tree<Number>(0);  // Note autoboxing
t.addBranch(new Tree<Integer>(1));     // int 1 autoboxed to Integer
```

If we query the branch with the `getBranch( )` method, the value type of the returned branch is unknown, and we must use a wildcard to express this. The next two lines are legal, but the third is not:

```
Tree<? extends Number> b = t.getBranch(0);
Tree<?> b2 = t.getBranch(0);
Tree<Number> b3 = t.getBranch(0);  // compilation error
```

When we query a branch like this, we don't know the precise type of the value, but we do still have an upper bound on the value type, so we can do this:

```
Tree<? extends Number> b = t.getBranch(0);
Number value = b.getValue();
```

What we cannot do, however, is set the value of the branch, or add a new branch to that branch. As explained earlier in the chapter, the existence of the upper bound does not change the fact that the value type is unknown. The compiler does not have enough information to allow us to safely pass a value to `setValue()` or a new branch (which includes a value type) to `addBranch()`. Both of these lines of code are illegal:

```
b.setValue(3.0); // Illegal, value type is unknown
b.addBranch(new Tree<Double>(Math.PI));
```

This example has illustrated a typical trade-off in the design of a generic type: using a bounded wildcard made the data structure more flexible but reduced our ability to safely use some of its methods. Whether or not this was a good design is probably a matter of context. In general, generic types are more difficult to design well. Fortunately, most of us will use the preexisting generic types in the `java.util` package much more frequently than we will have to create our own.

4.1.4.3. Generic methods

As noted earlier, the type variables of a generic type can be used only in the instance members of the type, not in the static members. Like instance methods, however, static methods can use wildcards. And although static methods cannot use the type variables of their containing class, they can declare their own type variables. When a method declares its own type variable, it is called a *generic method*.

Here is a static method that could be added to the `Tree` class. It is not a generic method but uses a bounded wildcard much like the `sumList()` method we saw earlier in the chapter:

```
/** Recursively compute the sum of the values of all nodes on the tree */
public static double sum(Tree<? extends Number> t) {
    double total = t.value.doubleValue();
    for(Tree<? extends Number> b : t.branches) total += sum(b);
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    return total;
}
```

This method could also be rewritten as a generic method by declaring a type variable to express the upper bound imposed by the wildcard:

```
public static <N extends Number> double sum(Tree<N> t) {
    N value = t.value;
    double total = value.doubleValue();
    for(Tree<? extends N> b : t.branches) total += sum(b);
    return total;
}
```

The generic version of `sum()` is no simpler than the wildcard version and the declaration of the type variable does not gain us anything. In a case like this, the wildcard solution is typically preferred over the generic solution. Generic methods are required where a single type variable is used to express a relationship between two parameters or between a parameter and a return value. The following method is an example:

```
// This method returns the largest of two trees, where tree size
// is computed by the sum() method.  The type variable ensures that
// both trees have the same value type and that both can be passed to sum().
public static <N extends Number> Tree<N> max(Tree<N> t, Tree<N> u) {
    double ts = sum(t);
    double us = sum(u);
    if (ts > us) return t;
    else return u;
}
```

This method uses the type variable `N` to express the constraint that both arguments and the return value have the same type parameter and that that type parameter is `Number` or a subclass.

It could be argued that constraining both arguments to have the same value type is too restrictive and that we should be allowed to call the `max( )` method on a `Tree<Integer>` and a `Tree<Double>`. One way to express this is to use two unrelated type variables to represent the two unrelated value types. Note, however, that we cannot use either variable in the return type of the method and must use a wildcard there:

```
public static <N extends Number, M extends Number>
    Tree<? extends Number> max(Tree<N> t, Tree<M> u) {...}
```

Since the two type variables `N` and `M` have no relation to each other, and since each is used in only a single place in the signature, they offer no advantage over bounded wildcards. The method is better written this way:

```
public static Tree<? extends Number> max(Tree<? extends Number> t,
                                         Tree<? extends Number> u) {...}
```

All the examples of generic methods shown here have been `static` methods. This is not a requirement: instance methods can declare their own type variables as well.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

4.1.4.4. Invoking generic methods

When you use a generic type, you must specify the actual type parameters to be substituted for its type variables. The same is not generally true for generic methods: the compiler can almost always figure out the correct parameterization of a generic method based on the arguments you pass to the method. Consider the `max()` method defined above, for instance:

```
public static <N extends Number> Tree<N> max(Tree<N> t, Tree<N> u) {...}
```

You need not specify `N` when you invoke this method because `N` is implicitly specified in the values of the method arguments *`t`* and *`u`*. In the following code, for example, the compiler determines that `N` is `Integer`:

```
Tree<Integer> x = new Tree<Integer>(1);
Tree<Integer> y = new Tree<Integer>(2);
Tree<Integer> z = Tree.max(x, y);
```

The process the compiler uses to determine the type parameters for a generic method is called *type inference*. Type inference is relatively intuitive to understand, but the actual algorithm the compiler must use is surprisingly complex and is well beyond the scope of this book. Complete details are in Chapter 15 of *The Java Language Specification, Third Edition*.

Let's look at a slightly more complex version of type inference. Consider this method:

```
public class Util {
    /** Set all elements of a to the value v; return a. */
    public static <T> T[] fill(T[] a, T v) {
        for(int i = 0; i < a.length; i++) a[i] = v;
        return a;
    }
}
```

Here are two invocations of the method:

```
Boolean[] booleans = Util.fill(new Boolean[100], Boolean.TRUE);
Object o = Util.fill(new Number[5], new Integer(42));
```

In the first invocation, the compiler can easily determine that `T` is `Boolean`. In the second invocation, the compiler determines that `T` is `Number`.

In very rare circumstances you may need to explicitly specify the type parameters for a generic method. This is sometimes necessary, for example, when a generic method expects no arguments. Consider the `java.util.Collections.emptySet( )` method: it returns a set with no elements, but unlike the `Collections.singleton( )` method (you can look these up in the reference section), it takes no arguments that would specify

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

the type parameter for the returned set. You can specify the type parameter explicitly by placing it in angle brackets *before* the method name:

```
Set<String> empty = Collections.<String>emptySet();
```

Type parameters cannot be used with an unqualified method name: they must follow a dot or come after the keyword `new` or before the keyword `this` or `super` used in a constructor.

It turns out that if you assign the return value of `Collections.emptySet()` to a variable, as we did above the type inference mechanism is able to infer the type parameter based on the variable type. Although the explicit type parameter specification in the code above can be a helpful clarification, it is not necessary and the line could be rewritten as:

```
Set<String> empty = Collections.emptySet();
```

An explicit type parameter is necessary when you use the return value of the `emptySet ( )` method within a method invocation expression. For example, suppose you want to call a method named `printWords( )` that expects a single argument of type `Set<String>`. If you want to pass an empty set to this method, you could use this code:

```
printWords(Collections.<String>emptySet());
```

In this case, the explicit specification of the type parameter `String` is required.

4.1.4.5. Generic methods and arrays

Earlier in the chapter we saw that the compiler does not allow you to create an array whose type is parameterized. This is not, however, a restriction on all uses of arrays with generics. Consider the `Util.fill()` method defined above, for example. Its first argument and its return value are both of type `T[]`. The body of the method does not have to create an array whose element type is `T`, so the method is perfectly legal.

If you write a method that uses varargs (see Section 2.6.4 in Chapter 2) and a type variable, remember that invoking a varargs method performs an implicit array creation. Consider this method:

```
/** Return the largest of the specified values or null if there are none */
public static <T extends Comparable<T>> T max(T... values) { ...  }
```

You can invoke this method with parameters of type `Integer` because the compiler can insert the necessary array creation code for you when you call it. But you cannot call the method if you've cast the same arguments to be type `Comparable<Integer>` because it is not legal to create an array of type `Comparable<Integer>[ ]`.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

4.1.4.6. Parameterized exceptions

Exceptions are thrown and caught at runtime, and there is no way for the compiler to perform type checking to ensure that an exception of unknown origin matches type parameters specified in a `catch` clause. For this reason, `catch` clauses may not include type variables or wildcards. Since it is not possible to catch an exception at runtime with compile-time type parameters intact, you are not allowed to make any subclass of `Throwable` generic. Parameterized exceptions are simply not allowed.

You can, however, use a type variable in the `throws` clause of a method signature. Consider this code, for example:

```
public interface Command<X extends Exception> {
    public void doit(String arg) throws X;
}
```

This interface represents a "command": a block of code with a single string argument and no return value. The code may throw an exception represented by the type parameter `X`. Here is an example that uses a parameterization of this interface:

```
Command<IOException> save = new Command<IOException>() {
    public void doit(String filename) throws IOException {
        PrintWriter out = new PrintWriter(new FileWriter(filename));
        out.println("hello world");
        out.close();
    }
};

try { save.doit("/tmp/foo");  }
catch(IOException e) { System.out.println(e); }
```

4.1.5. Generics Case Study: Comparable and Enum

The new generics features in Java 5.0 are used in the Java 5.0 APIs, most notably in `java.util` but also in `java.lang`, `java.lang.reflect`, and `java.util.concurrent`. These APIs were carefully created or reviewed by the inventors of generic types, and we can learn a lot about the good design of generic types and methods through the study of these APIs.

The generic types of `java.util` are relatively easy: for the most part they are collections classes, and type variables are used to represent the element type of the collection. Several important generic types in `java.lang` are more difficult. They are not collections, and it is not immediately apparent why they have been made generic. Studying these difficult generic types gives us a deeper understanding of how generics work and introduces some concepts that we have not yet covered in this chapter. Specifically, we'll examine the `Comparable` interface and the `Enum` class (the supertype of enumerated types, described later in this chapter) and will learn about an important but infrequently used feature of generics known as lower-bounded wildcards.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

In Java 5.0, the `Comparable` interface has been made generic, with a type variable that specifies what a class is comparable to. Most classes that implement `Comparable` implement it on themselves. Consider `Integer`:

```
public final class Integer extends Number implements Comparable<Integer>
```

The raw `Comparable` interface is problematic from a type-safety standpoint. It is possible to have two `Comparable` objects that cannot be meaningfully compared to each other. Prior to Java 5.0, the nongeneric `Comparable` interface was useful but not fully satisfactory. The generic version of this interface, however, captures exactly the information we want: it tells us that a type is comparable and tells us what we can compare it to.

Now consider subclasses of comparable classes. `Integer` is `final` and cannot be subclassed, so let's look at `java.math.BigInteger` instead:

```
public class BigInteger extends Number implements Comparable<BigInteger>
```

If we implement a `BiggerInteger` subclass of `BigInteger`, it inherits the `Comparable` interface from its superclass. But note that it inherits `Comparable<BigInteger>` and not `Comparable<BiggerInteger>`. This means that `BigInteger` and `BiggerInteger` objects are mutually comparable, which is usually a good thing. `BiggerInteger` can override the `compareTo( )` method of its superclass, but it is not allowed to implement a different parameterization of `Comparable`. That is, `BiggerInteger` cannot both extend `BigInteger` and implement `Comparable<BiggerInteger>`. (In general, a class is not allowed to implement two different parameterizations of the same interface: we cannot define a type that implements both `Comparable<Integer>` and `Comparable<String>`, for example.)

When you're working with comparable objects (as you do when writing sorting algorithms, for example), remember two things. First, it is not sufficient to use `Comparable` as a raw type: for type safety, you must also specify what it is comparable to. Second, types are not always comparable to themselves: sometimes they're comparable to one of their ancestors. To make this concrete, consider the `java.util.Collections.max()` method:

```
public static <T extends Comparable<? super T>> T max(Collection<? extends T> c)
```

This is a long, complex generic method signature. Let's walk through it:

- The method has a type variable `T` with complicated bounds that we'll return to later.
- The method returns a value of type T.
- The name of the method is `max( )`.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

- The method's argument is a Collection. The element type of the collection is specified with a bounded wildcard. We don't know the exact type of the collection's elements, but we know that they have an upper bound of `T`. That is, we know that the elements of the collection are type `T` or a subclass of `T`. Any element of the collection could therefore be used as the return value of the method.

That much is relatively straightforward. We've seen upper-bounded wildcards elsewhere in this section. Now let's look again at the type variable declaration used by the `max( )` method:

```
<T extends Comparable<? super T>>
```

This says first that the type `T` must implement `Comparable`. (Generics syntax uses the keyword `extends` for all type variable bounds, whether classes or interfaces.) This is expected since the purpose of the method is to find the "maximum" object in a collection. But look at the parameterization of the `Comparable` interface. This is a wildcard, but it is bounded with the keyword `super` instead of the keyword `extends`. This is a lower-bounded wildcard. `? extends T` is the familiar upper bound: it means `T` or a subclass. `? super T` is less commonly used: it means `T` or a superclass.

To summarize, then, the type variable declaration states "`T` is a type that is comparable to itself or to some superclass of itself." The `Collections.min()` and `Collections.binarySearch( )` methods have similar signatures.

For other examples of lower-bounded wildcards (that have nothing to do with `Comparable`), consider the `addAll()`, `copy( )`, and `fill()` methods of `Collections`. Here is the signature for `addAll()`:

```
public static <T> boolean addAll(Collection<? super T> c, T... a)
```

This is a varargs method that accepts any number of arguments of type `T` and passes them as a `T[ ]` named *`a`*. It adds all the elements of *`a`* to the collection *`c`*. The element type of the collection is unknown but has a lower bound: the elements are all of type `T` or a superclass of `T`. Whatever the type is, we are assured that the elements of the array are instances of that type, and so it is always legal to add those array elements to the collection.

Recall from our earlier discussion of upper-bounded wildcards that if you have a collection whose element type is an upper-bounded wildcard, it is effectively read-only. Consider `List<? extends Serializable>`. We know that all elements are `Serializable`, so methods like `get()` return a value of type `Serializable`. The compiler won't let us call methods like `add()` because the actual element type of the list is unknown. You can't add

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

arbitrary serializable objects to the list because their implementing class may not be of the correct type.

Since upper-bounded wildcards result in read-only collections, you might expect lower-bounded wildcards to result in write-only collections. This isn't actually the case, however. Suppose we have a `List<? super Integer>`. The actual element type is unknown, but the only possibilities are `Integer` or its ancestors `Number` and `Object`. Whatever the actual type is, it is safe to add `Integer` objects (but not `Number` or `Object` objects) to the list. And, whatever the actual element type is, all elements of the list are instances of `Object`, so `List` methods like `get( )` return `Object` in this case.

Finally, let's turn our attention to the `java.lang.Enum` class. `Enum` serves as the supertype of all enumerated types (described later). It implements the `Comparable` interface but has a confusing generic signature:

```
public class Enum<E extends Enum<E>> implements Comparable<E>, Serializable
```

At first glance, the declaration of the type variable `E` appears circular. Take a closer look though: what this signature really says is that `Enum` must be parameterized by a type that is itself an `Enum`. The reason for this seemingly circular type variable declaration becomes apparent if we look at the `implements` clause of the signature. As we've seen, `Comparable` classes are usually defined to be comparable to themselves. And subclasses of those classes are comparable to their superclass instead. `Enum`, on the other hand, implements the `Comparable` interface not for itself but for a subclass `E` of itself!

4.2. Enumerated Types

In previous chapters, we've seen the `class` keyword used to define class types, and the `interface` keyword used to define interface types. This section introduces the `enum` keyword, which is used to define an enumerated type (informally called an enum). Enumerated types are new in Java 5.0, and the features described here cannot be used (although they can be partially simulated) prior to that release.

We begin with the basics: how to define and use an enumerated type, including common programming idioms involving enumerated types and values. Next, we discuss the more advanced features of enums and show how to simulate enums prior to Java 5.0.

4.2.1. Enumerated Types Basics

An *enumerated type* is a reference type with a finite (usually small) set of possible values, each of which is individually listed, or enumerated. Here is a simple enumerated type defined in Java:

```
public enum DownloadStatus { CONNECTING, READING, DONE, ERROR }
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Like `class` and `interface`, the `enum` keyword defines a new reference type. The single line of Java code above defines an enumerated type named `DownloadStatus`. The body of this type is simply a comma-separated list of the four values of the type. These values are like `static final` fields (which is why their names are capitalized), and you refer to them with names like `DownloadStatus.CONNECTING`, `DownloadStatus.READING`, and so on. A variable of type `DownloadStatus` can be assigned one of these four values or `null` but nothing else. The values of an enumerated type are called *enumerated values* and are sometimes also referred to as *enum constants* .

It is possible to define more complex enumerated types than the one shown here, and we describe the complete `enum` syntax later in this chapter. For now, however, you can define simple, but very useful, enumerated types with this basic syntax.

4.2.1.1. Enumerated types are classes

Prior to the introduction of enumerated types in Java 5.0, the `DownloadStatus` values would probably have been implemented as integer constants with lines like the following in a class or interface:

```
public static final int CONNECTING = 1;
public static final int READING = 2;
public static final int DONE = 3;
public static final int ERROR = 4;
```

The use of integer constants has a number of shortcomings, the most important of which is its lack of type safety. If a method expects a download status constant value, for example, no error checking prevents me from passing an illegal value. The compiler can't tell me that I've used the constant `UploadStatus.DONE` when I should have used `DownloadStatus.DONE`.

Fortunately, enumerated types in Java are not simple integer constants. The type defined by an `enum` keyword is actually a class and its enumerated values are instances of that class. This provides type safety: if I try to pass a `DownloadStatus` value to a method that expects an `UploadStatus`, the compiler issues an error. Enumerated types do not have a public constructor, so a program cannot create a new undefined instance of the type. If a method expects a `DownloadStatus`, it can be confident that it will not be passed some unknown instance of the type.

If you are accustomed to writing code using integer constants instead of true enumerated types, you have probably already made a list of pragmatic advantages of integers over objects for enumerated values. Hold your judgment, however: the sections that follow illustrate common enumerated type programming idioms and demonstrate that anything

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

you can do with integer constants can be done elegantly, efficiently, and more safely with `enums`. First, however, we consider the basic features of all enumerated types.

4.2.1.2. Features of enumerated types

The following list describes the basic facts about enumerated types. These are the features of enums that you need to know to understand and use them effectively:

- Enumerated types have no public constructor. The only instances of an enumerated type are those declared by the enum.
- Enums are not `Cloneable`, so copies of the existing instances cannot be created.
- Enums implement `java.io.Serializable` so they can be serialized, but the Java serialization mechanism handles them specially to ensure that no new instances are ever created.
- Instances of an enumerated type are immutable: each enum value retains its identity. (We'll see later in this chapter that you can add your own fields and methods to an enumerated type, which means that you can create enumerated values that have mutable portions. This is not recommended, but does not affect the basic identity of each value.)
- Instances of an enumerated type are stored in `public static final` fields of the type itself. Because these fields are `final`, they cannot be overwritten with inappropriate values: you can't assign the `DownloadStatus.ERROR` value to the `DownloadStatus.DONE` field, for example.
- By convention, the values of enumerated types are written using all capital letters, just as other `static final` fields are.
- Because there is a strictly limited set of distinct enumerated values, it is always safe to compare enum values using the `==` operator instead of calling the `equals()` method.
- Enumerated types do have a working `equals( )` method, however. The method uses `==` internally and is `final` so that it cannot be overridden. This working `equals( )` method allows enumerated values to be used as members of collections such as `Set`, `List`, and `Map`.
- Enumerated types have a working `hashCode()` method consistent with their `equals( )` method. Like `equals()`, `hashCode( )` is `final`. It allows enumerated values to be used with classes like `java.util.HashMap`.
- Enumerated types implement `java.lang.Comparable`, and the `compareTo()` method orders enumerated values in the order in which they appear in the `enum` declaration.
- Enumerated types include a working `toString( )` method that returns the name of the enumerated value. For example, `DownloadStatus.DONE.toString( )` returns the string "`DONE`" by default. This method is not `final`, and enum types can provide a custom implementation if they choose.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

- Enumerated types provide a static `valueOf( )` method that does the opposite of the default `toString( )` method. For example, `DownloadStatus.valueOf ("DONE")` would return `DownloadStatus.DONE`.
- Enumerated types define a `final` instance method named `ordinal()` that returns an integer for each enumerated value. The ordinal of an enumerated value represents its position (starting at zero) in the list of value names in the `enum` declaration. You do not typically need to use the `ordinal( )` method, but it is used by a number of enum-related facilities, as described later in the chapter.
- Each enumerated type defines a static method named `values( )` that returns an array of enumerated values of that type. This array contains the complete set of values, in the order they were declared, and is useful for iterating through the complete set of possible values. Because arrays are mutable, the `values( )` method always returns a newly created and initialized array.
- Enumerated types are subclasses of `java.lang.Enum`, which is new in Java 5.0. (`Enum` is not itself an enumerated type.) You cannot produce an enumerated type by manually extending the `Enum` class, and it is a compilation error to attempt this. The only way to define an enumerated type is with the `enum` keyword.
- It is not possible to extend an enumerated type. Enumerated types are effectively `final`, but the `final` keyword is neither required nor permitted in their declarations. Because enums are effectively `final`, they may not be `abstract`. (We'll return to this point later in the chapter.)
- Like classes, enumerated types may implement interfaces. (We'll see how enumerated types may define methods later in the chapter.)

4.2.2. Using Enumerated Types

The following sections illustrate common idioms for working with enumerated types. They demonstrate the use of the `switch` statement with enumerated types and introduce the important new `EnumSet` and `EnumMap` collections.

4.2.2.1. Enums and the switch statement

In Java 1.4 and earlier, the `switch` statement works only with `int`, `short`, `char`, and `byte` values. Because enumerated types have a finite set of values, they are ideally suited for use with the `switch` statement, and this statement has been extended in Java 5.0 to support the use of enumerated types. If the compile-time type of the `switch` expression is an enumerated type, the `case` labels must all be unqualified names of instances of that type. The following hypothetical code shows a `switch` statement used with the `DownloadStatus` enumerated type.

```
DownloadStatus status = imageLoader.getStatus();
switch(status) {
case CONNECTING:
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
        imageLoader.waitForConnection();
        imageLoader.startReading();
        break;
    case READING:
        break;
    case DONE:
        return imageLoader.getImage();
    case ERROR:
        throw new IOException(imageLoader.getError());
    }
```

Note that the case labels are just the constant name: the syntax of the `switch` statement does not allow the class name `DownloadStatus` to appear here. The ability to omit the class name is very convenient since it would otherwise appear in every single `case`. However the *requirement* that the class name be omitted is surprising since (in the absence of an `import static` declaration) the class name *is* required in every other context.

If the `switch` expression (`status` in the code above) evaluates to `null`, a `NullPointerException` is thrown. It is not legal to use `null` as the value of a `case` label.

If you use the `switch` statement on an enumerated type and do not include either a `default`: label or a `case` label for each enumerated value, the compiler will most likely issue an `-Xlint` warning letting you know that you have not written code to handle all possible values of the enumerated type.[5] Even when you do write a `case` for each enumerated value, you may still want to include a `default`: clause; this covers the possibility that a new value is added to the enumerated type after your `switch` statement has been compiled. The following `default` clause, for example, could be added to the `switch` statement shown earlier:

[5] At the time of this writing, this warning is expected to appear in Java 5.1.

```
default: throw new AssertionError("Unexpected enumerated value: " + status);
```

4.2.2.2. EnumMap

A common programming technique when using integer constants instead of true enumerated values is to use those constants as array indexes. For example, if the `DownloadStatus` values are defined as integers between 0 and 3, we can write code like this:

```
String[] statusLineMessages = new String[] {
    "Connecting...",   // CONNECTING
    "Loading...",      // READING
    "Done.",           // DONE
    "Download Failed." // ERROR
};

int status = getStatus();
String message = statusLineMessages[status];
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

In the big picture, this technique creates a mapping from enumerated integer constants to strings. We can't use Java's enumerated values as array indexes, but we can use them as keys in a `java.util.Map`. Because this is a common thing to do, Java 5.0 defines a new `java.util.EnumMap` class that is optimized for exactly this case. `EnumMap` requires an enumerated type as its key, and, relying on the fact the number of possible keys is finite, it uses an array to hold the corresponding values. This implementation means that `EnumMap` is more efficient than `HashMap`. The `EnumMap` equivalent of the code above is:

```
EnumMap<DownloadStatus,String> messages =
    new EnumMap<DownloadStatus,String>(DownloadStatus.class);
messages.put(DownloadStatus.CONNECTING, "Connecting...");
messages.put(DownloadStatus.READING,    "Loading...");
messages.put(DownloadStatus.DONE,       "Done.");
messages.put(DownloadStatus.ERROR,      "Download Failed.");

DownloadStatus status = getStatus();
String message = messages.get(status);
```

Like other collection classes in Java 5.0, `EnumMap` is a generic type that accepts type parameters.

The use of an `EnumMap` to associate a value with each instance of an enumerated type is appropriate when you're working with an enum defined elsewhere. If you defined the enum value yourself, you can create the necessary associations as part of the `enum` definition itself. We'll see how to do this later in the chapter.

4.2.2.3. EnumSet

Another common programming idiom when using integer-based constants instead of an enumerated type is to define all the constants as powers of two so that a set of those constants can be compactly represented as bit-flags in an integer. Consider the following flags that describe options that can apply to an American-style espresso drink:

```
public static final int SHORT      = 0x01;  // 8 ounces
public static final int TALL       = 0x02;  // 12 ounces
public static final int GRANDE     = 0x04;  // 16 ounces
public static final int DOUBLE     = 0x08;  // 2 shots of espresso
public static final int SKINNY     = 0x10;  // made with nonfat milk
public static final int WITH_ROOM  = 0x20;  // leave room for cream
public static final int SPLIT_SHOT = 0x40;  // half decaffeinated
public static final int DECAF      = 0x80;  // fully decaffeinated
```

These power-of-two constants can be combined with the bitwise OR operator (`|`) to create a compact set of constants that is easy to work with:

```
int drinkflags = DOUBLE | SHORT | WITH_ROOM;
```

The bitwise AND operator (`&`) can be used to test for the presence or absence of bits:

```
boolean isBig = (drinkflags & (TALL | GRANDE)) != 0;
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

If we step back from the binary representation of these bit flags and the boolean operators that manipulate them, we can see that integer bit flags are simply compact sets of values. For reference types such as Java's enumerated values, we can use a `java.util.Set` instead. Since this is an important and common thing to do with enumerated values, Java 5.0 provides the special-purpose `java.util.EnumSet` class. Like `EnumMap`, `EnumSet` is optimized for enumerated types. It requires that its members be values of the same enumerated type and uses a compact and fast representation of the set based on bit flags that correspond to the `ordinal()` of each enumerated value.

The espresso drink code above could be rewritten as follows using an `enum` and `EnumSet`:

```
public enum DrinkFlags {
    SHORT, TALL, GRANDE, DOUBLE, SKINNY, WITH_ROOM, SPLIT_SHOT, DECAF
}

EnumSet<DrinkFlags> drinkflags =
    EnumSet.of(DrinkFlags.DOUBLE, DrinkFlags.SHORT, DrinkFlags.WITH_ROOM);

boolean isbig =
    drinkflags.contains(DrinkFlags.TALL) ||
    drinkflags.contains(DrinkFlags.GRANDE);
```

Note that the code above can be made as compact as the integer-based code with a simple static import:

```
// Import all static DrinkFlag enum constants
import static com.davidflanagan.coffee.DrinkFlags.*;
```

See Section 2.10 in Chapter 2 for details on the `import static` declaration.

`EnumSet` defines a number of useful factory methods for initializing sets of enumerated values. The `of()` method shown above is overloaded: several versions of the method take different fixed numbers of arguments. A varargs (see Chapter 2) form that can accept any number of arguments is also defined. Here are some other ways that you can use `of()` and related `EnumSet` factories:

```
// Make the following examples fit on the page better
import static com.davidflanagan.coffee.DrinkFlags.*;

// We can remove individual members or sets of members from a set.
// Start with a set that includes all enumerated values, then remove a subset:
EnumSet<DrinkFlags> fullCaffeine = EnumSet.allOf(DrinkFlags.class);
fullCaffeine.removeAll(EnumSet.of(DECAF, SPLIT_SHOT));

// Here's another technique to achieve the same result:
EnumSet<DrinkFlags> fullCaffeine =
    EnumSet.complementOf(EnumSet.of(DECAF,SPLIT_SHOT));

// Here's an empty set if you ever need one
// Note that since we don't specify a value, we must specify the element type
EnumSet<DrinkFlags> plainDrink = EnumSet.noneOf(DrinkFlags.class);

// You can also easily describe a contiguous subset of values:
EnumSet<DrinkFlags> drinkSizes = EnumSet.range(SHORT, GRANDE);
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
// EnumSet is Iterable, and its iterator returns values in ordinal() order,
// so it is easy to loop through the elements of an EnumSet.
for(DrinkFlag size : drinkSizes) System.out.println(size);
```

The example code shown here demonstrates the use and capabilities of the `EnumSet` class. Note, however, that an `EnumSet<DrinkFlags>` is not really an appropriate representation for the description of an espresso drink. An `EnumSet<DrinkFlags>` might be overspecified, including both `SHORT` and `GRANDE`, for example, or it might be underspecified and include no drink size at all.

At the root, the problem is that the `DrinkFlag` type is a naive translation of the integer bit flags we began this section with. A better and more complete representation is captured by the following interface, which requires one value from each of five different enumerated types and a set of values from a sixth enum. The enums are defined as *nested types* within the interface itself (see Chapter 3). This example highlights the type safety provided by enumerated types. It is not possible (as it would be with integer constants) to specify a drink strength where a drink size is required, for example.

```
public interface Espresso {
    enum Drink { LATTE, MOCHA, AMERICANO, CAPPUCCINO, ESPRESSO }
    enum Size { SHORT, TALL, GRANDE }
    enum Strength { SINGLE, DOUBLE, TRIPLE, QUAD }
    enum Milk { SKINNY, ONE_PERCENT, TWO_PERCENT, WHOLE, SOY }
    enum Caffeine { REGULAR, SPLIT_SHOT, DECAF }
    enum Flags { WITH_ROOM, EXTRA_HOT, DRY }

    Drink getDrink();
    Size getSize();
    Strength getStrength();
    Milk getMilk();
    Caffeine getCaffeine();
    java.util.Set<Flags> getFlags();
}
```

4.2.3. Advanced Enum Syntax

The examples shown so far have all used the simplest `enum` syntax in which the body of the enum simply consists of a comma-separated list of value names. The full `enum` syntax actually provides quite a bit more power and flexibility:

- You can define your own fields, methods, and constructors for the enumerated type.
- If you define one or more constructors, you can invoke a constructor for each enumerated value by following the value name with constructor arguments in parentheses.
- Although an `enum` may not `extend` anything, it may `implement` one or more interfaces.
- Most esoterically, individual enumerated values can have their own class bodies that override methods defined by the type.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Rather than formally specifying the syntax for each of these advanced `enum` declarations, we'll demonstrate the syntax in the examples that follow.

4.2.3.1. The class body of an enumerated type

Consider the type `Prefix`, defined below. It is an `enum` that includes a regular class body following the list of enumerated values. It defines two instance fields and accessor methods for those fields. It defines a custom constructor that initializes the instance field. Each named value of the enumerated type is followed by constructor arguments in parentheses:

```
public enum Prefix {
    // These are the values of this enumerated type.
    // Each one is followed by constructor arguments in parentheses.
    // The values are separated from each other by commas, and the
    // list of values is terminated with a semicolon to separate it from
    // the class body that follows.
    MILLI("m",    .001),
    CENTI("c",    .01),
    DECI("d",     .1),
    DECA("D",   10.0),
    HECTA("h", 100.0),
    KILO("k", 1000.0);  // Note semicolon


    // This is the constructor invoked for each value above.
    Prefix(String abbrev, double multiplier) {
        this.abbrev = abbrev;
        this.multiplier = multiplier;
    }

    // These are the private fields set by the constructor
    private String abbrev;
    private double multiplier;

    // These are accessor methods for the fields.  They are instance methods
    // of each value of the enumerated type.
    public String abbrev() { return abbrev; }
    public double multiplier() { return multiplier; }
}
```

Note that `enum` syntax requires a semicolon after the last enumerated value if that value is followed by a class body. This semicolon may be omitted in the simple case where there is no class body. It is also worth noting that `enum` syntax allows a comma following the last enumerated value. A trailing comma looks somewhat odd but prevents syntax errors if in the future you add new enumerated values or rearrange existing ones.

4.2.3.2. Implementing an interface

An `enum` cannot be declared to `extend` a class or enumerated type. It is perfectly legal, however, for an enumerated type to `implement` one or more interfaces. Suppose, for example, that you defined a new enumerated type `Unit` with an `abbrev( )` method like `Prefix` has. In this case, you might define an interface `Abbrevable` for any objects that have abbreviations. Your code might look like this:

```
public interface Abbrevable {
    String abbrev();
}
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
public enum Prefix implements Abbrevable {
    // the body of this enum type remains the same as above.
}
```

4.2.3.3. Value-specific class bodies

In addition to defining a class body for the enumerated type itself, you can also provide a class body for individual enumerated values within the type. We've seen above that we can add fields to an enumerated type and use a constructor to initialize those fields. This gives us value-specific data. The ability to define class bodies for each enumerated value means that we can write methods for each one: this gives us value-specific *behavior*. Value-specific behavior is useful when defining an enumerated type that represents an operator in an expression parser or an opcode in a virtual machine of some sort. The `Operator.ADD` constant might have a `compute()` method that behaves differently than the `Operator.SUBTRACT` constant, for example.

To define a class body for an individual enumerated value, simply follow the value name and its constructor arguments with the class body in curly braces. Individual values must still be separated from each other with commas, and the last value in the list must be separated from the type's class body with a semicolon: it can be easy to forget about this required punctuation with the presence of curly braces for class and method bodies.

Each value-specific class body you write results in the creation of an anonymous subclass of the enumerated type and makes the enumerated value a singleton instance of that anonymous subclass. (Enumerated types can not be extended, but they are not strictly `final` in the sense that `final` classes are since they can have these anonymous subclasses.) Because these subclasses are anonymous, you cannot refer to them in your code: the compile-time type of each enumerated value is the enumerated type, not the anonymous subclass specific to that value. Therefore, the only useful thing you can do in value-specific class bodies is override methods defined by the type itself. If you define a new public field or method, you will not be able to refer to or invoke it. (It is perfectly legitimate, of course, to define helper methods or fields that you invoke or use from the overriding methods.)

A common pattern is to define default behavior in a method of the type-specific class body. Then, each enumerated value that requires behavior other than the default can override that method in its value-specific class body. A very useful variant of this pattern is to declare the method in the type-specific class body `abstract` and to define a value-specific implementation of the method for every enumerated value. If the type-specific method is `abstract`, the compiler forces you to implement that method for every enumerated value in the type: it is not possible to accidentally omit an implementation. Note that even though the type-specific class body contains an `abstract` method, the enumerated type as a whole is not `abstract` (and may not be declared `abstract`) since each value-specific class body implements the method.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

The following code is an excerpt from a larger example that uses an enumerated type to represent the opcodes of a simulated stack-based CPU. The `Opcode` enumerated type defines an `abstract` method `perform()`, which is then implemented by the class body of each value of the type. The type includes a constructor to illustrate the full syntax for each enumerated value: name, constructor arguments, and class body. `enum` syntax requires the enumerated values and their class bodies to appear first. The code is easiest to understand, however, if you skip past the values and read the type-specific class body first:

```
// These are the opcodes that our stack machine can execute.
public enum Opcode {
    // Push the single operand onto the stack
    PUSH(1) {
        public void perform(StackMachine machine, int[] operands) {
            machine.push(operands[0]);
        }
    },   // Remember to separate enum values with commas

    // Add the top two values on the stack and push the result
    ADD(0) {
        public void perform(StackMachine machine, int[] operands) {
            machine.push(machine.pop() + machine.pop());
        }
    },

    /* Other opcode values have been omitted for brevity */

    // Branch if Equal to Zero
    BEZ(1) {
        public void perform(StackMachine machine, int[] operands) {
            if (machine.top() == 0) machine.setPC(operands[0]);
        }
    };    // Remember the required semicolon before the class body


    // This is the constructor for the type.
    Opcode(int numOperands) { this.numOperands = numOperands; }

    int numOperands;  // how many integer operands does it expect?

    // Each opcode constant must implement this abstract method in a
    // value-specific class body to perform the operation it represents.
    public abstract void perform(StackMachine machine, int[] operands);
}
```

4.2.3.3.1. When to use value-specific class bodies

Value-specific class bodies are an extremely powerful language feature when each enumerated value must perform a unique computation of some sort. Keep in mind, however, that value-specific class bodies are an advanced feature that is not commonly used and may be confusing to less experienced programmers. Before you decide to use this feature, be sure that it is necessary.

Before using value-specific class bodies, ensure that your design is neither too simple nor too complex for the feature. First, check that you do indeed require value-specific behavior and not simply value-specific data. Value-specific data can be encoded in constructor arguments as was shown in the `Prefix` example earlier. It would be unnecessary and

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

inappropriate to rewrite that example to use value-specific versions of the `abbrev( )` method, for example.

Next, think about whether an enumerated type is sufficient for your needs. If your design requires value-specific methods with complex implementations or requires more than a few methods for each value, you may find it unwieldy to code everything within a single type. Instead, consider defining your own custom type hierarchy using traditional `class` and `interface` declarations and whatever singleton instances are necessary.

If value-specific behavior is indeed required within the framework of an enumerated type, value-specific class bodies are appropriate. Whether value-specific bodies are truly elegant or simply confusing is a matter of opinion, and some programmers prefer to avoid them when possible. An alternative that appeals to some is to encode the value-specific behavior in a type-specific method that uses a `switch` statement to treat each value as a separate `case`. The `compute( )` method of the following `enum` is an example. The simplicity of this enumerated type makes a `switch` statement a compelling alternative to value-specific class bodies:

```
public enum ArithmeticOperator {
    // The enumerated values
    ADD, SUBTRACT, MULTIPLY, DIVIDE;

    // Value-specific behavior using a switch statement
    public double compute(double x, double y) {
        switch(this) {
        case ADD:       return x + y;
        case SUBTRACT:  return x - y;
        case MULTIPLY:  return x * y;
        case DIVIDE:    return x / y;
        default: throw new AssertionError(this);
        }
    }

    // Test case for using this enum
    public static void main(String args[]) {
        double x = Double.parseDouble(args[0]);
        double y = Double.parseDouble(args[1]);
        for(ArithmeticOperator op : ArithmeticOperator.values())
            System.out.printf("%f %s %f = %f%n", x, op, y, op.compute(x,y));
    }
}
```

A shortcoming to the `switch` approach is that each time you add a new enumerated value, you must remember to add a corresponding `case` to the `switch` statement. And if there is more than one method that uses a `switch` statement, you'll have to maintain their `switch` statements in parallel. Forgetting to implement value-specific behavior using a `switch` statement leads to a runtime `AssertionError`. With a value-specific class body overriding an `abstract` method in the type-specific class body, the same omission leads to a compilation error and can be corrected sooner.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

The performance of value-specific methods and `switch` statements in a type-specific method are quite similar. The overhead of virtual method invocation in one case is balanced by the overhead of the `switch` statement in the other. Value-specific class bodies result in the generation of additional class files, each of which has overhead in terms of storage space and loading time.

4.2.3.4. Restrictions on enum types

Java places a few restrictions on the code that can appear in an enumerated type. You won't encounter these restrictions that often in practice, but you should still be aware of them.

When you define an enumerated type, the compiler does a lot of work behind the scenes: it creates a class that extends `java.lang.Enum` and it generates the `values()` and `valueOf()` methods as well as the static fields that hold the enumerated values. If you include a class body for the type, you should not include members whose names conflict with the automatically generated members or with the `final` methods inherited from `Enum`.

`enum` types may not be declared `final`. Enumerated types are effectively final, and the compiler does not allow you to extend an `enum`. The class file generated for an `enum` is not technically declared `final` if the enum contains value-specific class bodies, however.

Types in Java may not be both `final` and `abstract`. Since enumerated types are effectively `final`, they may not be declared `abstract`. If the type-specific class body of an `enum` declaration contains an `abstract` method, the compiler requires that each enum value have a value-specific class body that includes an implementation of that `abstract` method. Considered as a self-contained whole, the enumerated type defined this way is not `abstract`.

The constructor, instance field initializers, and instance initializer blocks of an enumerated type are subject to a sweeping but obscure restriction: they may not use the static fields of the type (including the enumerated values themselves). The reason for this is that static initialization of enumerated types (and of all types) proceeds from top to bottom. The enumerated values are static fields that appear at the top of the type and are initialized first. Since they are self-typed fields, they invoke the constructor and any other instance initializer code of the type. This means that the instance initialization code is invoked before the static initialization of the class is complete. Since the static fields have not been initialized yet, the compiler does not allow them to be used. The only exception is static fields whose values are compile-time constant expressions (such as integers and strings) that the compiler resolves.

If you define a constructor for an enumerated type, it may not use the `super( )` keyword to invoke the superclass constructor. This is because the compiler automatically inserts

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

hidden `name` and `ordinal` arguments into any constructor you define. If you define more than one constructor for the type, it is okay to use `this()` to invoke one constructor from the other. Remember that the class bodies of individual enumerated values (if you define any) are anonymous, which means that they cannot have any constructors at all.

4.2.4. The Typesafe Enum Pattern

For a deeper understanding of how the `enum` keyword works, or to be able to simulate enumerated types prior to Java 5.0, it is useful to understand the *Typesafe Enum Pattern*. This pattern is described definitively by Joshua Bloch[6] in his book *Effective Java Programming Language Guide* (Addison Wesley); we do not cover all the nuances here.

[6] Josh was cochair of the the JSR 201 committee that developed many of the new language features of Java 5.0. He is the creator of and the driving force behind enumerated types.

If you want to use the enumerated type `Prefix` (from earlier in the chapter) prior to Java 5.0, you could approximate it with a class like the following one. Note, however, that instances of this class won't work with the `switch` statement or with the `EnumSet` and `EnumMap` classes. Also, the code shown here does not include the `values()` or `valueOf()` methods that the compiler generates automatically for true `enum` types. A class like this does not have special serialization support like an `enum` type does, so if you make it `Serializable`, you must provide a `readResolve()` method to prevent deserialization from creating multiple instances of the enumerated values.

```
public final class Prefix {
    // These are the self-typed constants
    public static final Prefix MILLI = new Prefix("m",     .001);
    public static final Prefix CENTI = new Prefix("c",     .01);
    public static final Prefix DECI  = new Prefix("d",     .1);
    public static final Prefix DECA  = new Prefix("D",   10.0);
    public static final Prefix HECTA = new Prefix("h",  100.0);
    public static final Prefix KILO  = new Prefix("k", 1000.0);

    // Keep the fields private so the instances are immutable
    private String name;
    private double multiplier;

    // The constructor is private so no instances can be created except
    // for the ones above.
    private Prefix(String name, double multiplier) {
        this.name = name;
        this.multiplier = multiplier;
    }

    // These accessor methods are public
    public String toString() { return name; }
    public double getMultiplier() { return multiplier; }
}
```

4.3. Annotations

Annotations provide a way to associate arbitrary information or *metadata* with program elements. Syntactically, annotations are used like modifiers and can be applied to the

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

declarations of packages, types, constructors, methods, fields, parameters, and local variables. The information stored in an annotation takes the form of *`name=value`* pairs, whose type is specified by the *annotation type*. The annotation type is a kind of interface that also serves to provide access to the annotation through the Java Reflection API.

Annotations can be used to associate any kind of information you want with a program element. The only fundamental rule is that an annotation cannot affect the way the program runs: the code must run identically even if you add or remove annotations. Another way to say this is that the Java interpreter ignores annotations (although it does make "runtime-visible" annotations available for reflective access through the Java Reflection API). Since the Java VM ignores annotations, an annotation type is not useful unless accompanied by a tool that can do something with the information stored in annotations of that type. In this chapter we'll cover standard annotation and meta-annotation types like `Override` and `Target`. The tool that accompanies these types is the Java compiler, which must process them in certain ways (as we'll describe later in this section).

It is easy to imagine any number of other uses for annotations.[7] A local variable might be annotated with a type named NonNull, as an assertion that the variable would never have a `null` value. An associated (hypothetical) code-analysis tool could then parse the code and attempt to verify the assertion. The JDK includes a tool named *apt* (for Annotation Processing Tool) that provides a framework for annotation processing tools: it scans source code for annotations and invokes specially written annotation processor classes that you provide. See Chapter 8 for more on *apt*. Annotations will probably find their widest use in enterprise programming where they may replace tools such as *XDoclet*, which processes metadata embedded in ad-hoc javadoc comments.

[7] We won't have to imagine these uses for long. At the time of this writing, JSR 250 is making its way through the Java Community Process to define a standard set of common annotations for J2SE and J2EE.

This section begins with an introduction to annotation-related terminology. We then cover the standard annotation types introduced in Java 5.0, annotations supported by *javac* that you can use in your programs right away. Next, we describe the syntax for writing arbitrary annotations and briefly cover the use of the Java Reflection API for querying annotations at runtime. At this point, we move on to more esoteric material on defining new annotation types, a task that few programmers will ever need to do. This final part of the chapter also discusses meta-annotations.

4.3.1. Annotation Concepts and Terminology

The key concept to understand about annotations is that an annotation simply associates information or metadata with a program element. Annotations *never affect the way a Java program runs*, but they may affect things like compiler warnings or the behavior of auxiliary tools such as documentation generators, stub generators, and so forth.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

The following terms are used frequently when discussing annotations. Of particular importance is the distinction between *annotation* and *annotation type*.

annotation

An *annotation* associates arbitrary information or metadata with a Java program element. Annotations use new syntax introduced in Java 5.0 and behave like modifiers such as `public` or `final`. Each annotation has a name and zero or more members. Each member has a name and a value, and it is these *`name=value`* pairs that carry the annotation's information.

annotation type

The name of an annotation as well as the names, types, and default values of its members are defined by the *annotation type*. An annotation type is essentially a Java interface with some restrictions on its members and some new syntax used in its declaration. When you query an annotation using the Java Reflection API, the returned value is an object that implements the annotation type interface and allows individual annotation members to be queried. Java 5.0 includes three standard annotation types in the `java.lang` package. We'll see these annotations in Section 4.3.2 later in this chapter.

annotation member

The *members* of an annotation are declared in an annotation type as no-argument methods. The method name and return type define the name and type of the member. A special `default` syntax allows the declaration of a default value for any annotation member. An annotation appearing on a program element includes *`name=value`* pairs that define values for all annotation members that do not have default values and may also include values that override the defaults of other members.

marker annotation

An annotation type that defines no members is called a *marker annotation*. An annotation of this type carries information simply by its presence or absence.

meta-annotation

A *meta-annotation* is an annotation applied to the declaration of an annotation type. Java 5.0 includes several standard meta-annotation types in the `java.lang.annotation` package. They are used to specify things like which program elements the annotation can be applied to.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

target

The *target* of an annotation is the program element that is annotated. Annotations can be applied to packages, types (classes, interfaces, enumerated types, and even annotation types), type members (methods, constructors, fields, and enumerated values), method parameters, and local variables (including loop variables and `catch` parameters). The declaration of an annotation type may include a *meta-annotation* that restricts the allowable targets for that type of annotation.

retention

The *retention* of an annotation specifies how long the information contained in the annotation is retained. Some annotations are discarded by the compiler and appear only in source code. Others are compiled into the class file. Of those that are compiled into the class file, some are ignored by the virtual machine, and others are read by the virtual machine when the class that contains them is loaded. The declaration of an annotation type can use a *meta-annotation* to specify the retention for annotations of that type. Annotations that are loaded by the VM are *runtime-visible* and can be queried by the reflective APIs of `java.lang.reflect`.

metadata

When discussing annotations, the term *metadata* commonly refers to the information carried by an annotation or to the annotation itself. Because this term is used in many different ways in computer programming literature, I have avoided using it in this chapter.

4.3.2. Using Standard Annotations

Java 5.0 defines three standard annotation types in the `java.lang` package. The following sections describe these annotation types and explain how to use them to annotate your code.

4.3.2.1. Override

`java.lang.Override` is a marker annotation type that can be used to annotate methods but no other program element. An annotation of this type serves as an assertion that the annotated method overrides a method of a superclass. If you use this annotation on a method that does not override a superclass method, the compiler issues a compilation error to alert you to this fact.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

This annotation is intended to address a common category of programming errors that result when you attempt to override a superclass method but get the method name or signature wrong. In this case, you may have overloaded the method name but not actually overridden the method, and your code never gets invoked.

To use this annotation type, simply include `@Override` in the modifiers of the desired method. By convention, `@Override` comes before other modifiers. Also by convention, there is no space between the `@` character and the name `Override`, even though it is technically allowed. Note that because the `java.lang` package is always automatically imported, you never need to include the package name to use this annotation type. Here is an example in which the `@Override` annotation is used on a method that fails to correctly override the `toString()` method of its superclass.

```
@Override
public String toSting() {   // Oops.  Note the misspelling here!
    // Simply put square brackets around our superclass's output
    return "[" + super.toString() + "]";
}
```

Without the annotation, the typo might go unnoticed and we'd have a puzzling bug: why isn't the `toString( )` method working correctly? But with the annotation, the compiler gives us the answer: the `toString()` method does not work as expected because it is not actually overridden.

Note that the `@Override` annotation applies only to methods that are intended to override a superclass method and not to methods that are intended to implement a method defined in an interface. The compiler already produces an error if you fail to correctly implement an interface method.

4.3.2.2. Deprecated

`java.lang.Deprecated` is a marker annotation that is similar to the `@deprecated` javadoc tag. (See Chapter 7 for details on writing Java documentation comments.) If you annotate a type or type member with `@Deprecated`, it tells the compiler that use of the annotated element is discouraged. If you use (or extend or override) a deprecated type or member from code that is not itself declared `@Deprecated`, the compiler issues a warning.

Note that the `@Deprecated` annotation type does not deprecate the `@deprecated` javadoc tag. The `@Deprecated` annotation is intended for the Java compiler. The javadoc tag, on the other hand, is intended for the *javadoc* tool and serves as documentation: it may include a description of why the program element has been deprecated and what it has been superseded by or replaced with.

In Java 5.0, the compiler continues to look for `@deprecated` javadoc tags and uses them to generate warnings as it always has. This behavior may be phased out, however, and you

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

should begin to use the `@Deprecated` annotation in addition to the `@deprecated` javadoc tag.

Here is an example that uses both the annotation and the javadoc tag:

```
/**
 * The Sony Betamax video cassette format.
 * @deprecated No one has players for this format any more.  Use VHS instead.
 */
@Deprecated public class Betamax { ... }
```

4.3.2.3. SuppressWarnings

The `@SuppressWarnings` annotation is used to selectively turn off compiler warnings for classes, methods, or field and variable initializers.[8] In Java 5.0, Sun's *javac* compiler has a powerful `-Xlint` option that causes it to issue warnings about "lint" in your program—code that is legal but is likely to represent a programming error. These warnings include the "unchecked warning" that appears when you use a generic collection class without specifying a value for its type parameters, for example, or the warning that appears if a `case` in a `switch` statement does not end with a `break`, `return`, or `throw` and allows control to "fall through" to the next case.

[8] The *javac* compiler did not yet support the `@SuppressWarnings` annotation when this chapter was written. Full support is expected in Java 5.1.

Typically, when you see one of these lint warnings from the compiler, you should investigate the code that caused it. If it truly represents an error, you then correct it. If it simply represents sloppy programming, you may be able to rewrite your code so that the warning is no longer necessary. For example, if the warning tells you that you have not covered all possible cases in a `switch` statement on an enumerated type, you can avoid the warning by adding a defensive `default` case to the `switch` statement, even if you are sure that it will never be invoked.

On the other hand, sometimes there is nothing you can do to avoid the error. For example, if you use a generic collection class in code that must interact with nongeneric legacy code, you cannot avoid an unchecked warning. This is where `@SuppressWarnings` comes in: add this annotation to the nearest relevant set of modifiers (typically on method modifiers) to tell the compiler that you're aware of the issue and that it should stop pestering you about it.

Unlike `Override` and `Deprecated`, `SuppressWarnings` is not a marker annotation. It has a single member named `value` whose type is `String[ ]`. The value of this member is the names of the warnings to be suppressed. The `SuppressWarnings` annotation does not define what warning names are allowed: this is an issue for compiler implementors. For the *javac* compiler, the warning names accepted by the `-Xlint` option are also legal for the `@SuppressWarnings` annotation. It is legal to specify any warning names you want: compilers ignore (but may warn about) warning names they do not recognize.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

So, to suppress warnings named `unchecked` and `fallthrough`, you could use an annotation that looks like the following. Annotation syntax follows the name of the annotation type with a parenthesized, comma-separated list of *`name=value`* pairs. In this case, the `SuppressWarnings` annotation type defines only a single member, so there is only a single pair within parentheses. Since the member value is an array, curly braces are used to delimit array elements:

```
@SuppressWarnings(value={"unchecked","fallthrough"})
public void lintTrap() { /* sloppy method body omitted */ }
```

We can abbreviate this annotation somewhat. When an annotation has a single member and that member is named "value", you are allowed (and encouraged) to omit the "value=" in the annotation. So the annotation above should be rewritten as:

```
@SuppressWarnings({"unchecked","fallthrough"})
```

Hopefully you will not often have more than one unresolvable lint warning in any particular method and will need to suppress only a single named warning. In this case, another annotation abbreviation is possible. When writing an array value that contains only a single member, you are allowed to omit the curly braces. In this case we might have an annotation like this:

```
@SuppressWarnings("unchecked")
```

4.3.3. Annotation Syntax

In the descriptions of the standard annotation types, we've seen the syntax for writing marker annotations and the syntax for writing single-member annotations, including the shortcut allowed when the single member is named "value" and the shortcut allowed when an array-typed member has only a single array element. This section describes the complete syntax for writing annotations.

An annotation consists of the `@` character followed by the name of the annotation type (which may include a package name) followed by a parenthesized, comma-separated list of *`name=value`* pairs for each of the members defined by the annotation type. Members may appear in any order and may be omitted if the annotation type defines a default value for that member. Each *`value`* must be a literal or compile-time constant, a nested annotation, or an array.

Near the end of this chapter, we define an annotation type named `Reviews` that has a single member that is an array of `@Review` annotations. The `Review` annotation type has three members: "reviewer" is a `String`, "comment" is an optional `String` with a default value, and "grade" is a value of the nested enumerated type `Review.Grade`. Assuming that the `Reviews` and `Review` types are properly imported, an annotation using these

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

types might look like this (note the use of nested annotations, enumerated types, and arrays in this annotation):

```
@Reviews({  // Single-value annotation, so "value=" is omitted here
    @Review(grade=Review.Grade.EXCELLENT,
            reviewer="df"),
    @Review(grade=Review.Grade.UNSATISFACTORY,
            reviewer="eg",
            comment="This method needs an @Override annotation")
})
```

Another important rule of annotation syntax is that no program element may have more than one instance of the same annotation. It is not legal, for example, to simply place multiple `@Review` annotations on a class. This is why the `@Reviews` annotation is defined to allow an array of `@Review` annotations.

4.3.3.1. Annotation member types and values

The values of annotation members must be non-`null` compile-time constant expressions that are assignment-compatible with the declared type of the member. Allowed member types are the primitive types, `String`, `Class`, enumerated types, annotation types, and arrays of any of the above types (but not an array of arrays). For example, the expressions `2*Math.PI` and `"hello"+"world"` are legal values for members of type `double` and `String`, respectively.

Near the end of the chapter, we define an annotation type named `UncheckedExceptions` whose sole member is an array of classes that extend `RuntimeException`. An annotation of this type might look like this:

```
@UncheckedExceptions({
    IllegalArgumentException.class, StringIndexOutOfBoundsException.class
})
```

4.3.3.2. Annotation targets

Annotations are most commonly placed on type definitions (such as classes) and their members (such as methods and fields). Annotations may also appear on packages, parameters, and local variables. This section provides more information about these less common annotation targets.

A package annotation appears before the `package` declaration in a file named *package-info.java*. This file should not contain any type declarations ("package-info" is not a legal Java identifier, so it cannot contain any public type definitions). Instead, it should contain an optional javadoc comment, zero or more annotations, and a `package` declaration. For example:

```
/**
 * This package holds my custom annotation types.
 */
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
@com.davidflanagan.annotations.Author("David Flanagan")
package com.davidflanagan.annotations;
```

When the *package-info.java* file is compiled, it produces a class file named *package-info.class* that contains a synthetic interface declaration. This interface has no members, and its name, `package-info`, is not a legal Java identifier, so it cannot be used in Java source code. It exists simply as a placeholder for package annotations with class or runtime retention.

Note that package annotations appear outside the scope of any `package` or `import` declaration. This means that package annotations should always include the package name of the annotation type (unless the package is `java.lang`).

Annotations on method parameters, catch clause parameters, and local variables simply appear as part of the modifier list for those program elements. The Java class file format has no provision for storing annotations on local variables or catch clause parameters, so those annotations always have source retention. Method parameter annotations can be retained in the class file, however, and may have class or runtime retention.

Finally, note that the syntax for enumerated type definitions does not allow any modifiers to be specified for enumerated values. It does, however, allow annotations on any of the values.

4.3.3.3. Annotations and defaults

Annotations must include a value for every member that does not have a default value defined by the annotation type. Annotations may, of course, include values for other members as well.

There is one important detail to understand about how default values are handled. Default values are stored in the class file of the annotation type and are not compiled into annotations themselves. If you modify an annotation type so that the default value of one of its members changes, that change affects all annotations of that type that do not specify an explicit value for that member. Already-compiled annotations are affected, even if they are never recompiled after the change to the type.

4.3.4. Annotations and Reflection

The Reflection API of `java.lang.reflect` has been extended in Java 5.0 to support reading of runtime-visible annotations. (Remember that an annotation is only visible at runtime if its annotation type is specified to have runtime retention, that is, if the annotation is both stored in the class file and read by the Java VM when the class file is loaded.) This section briefly covers the new reflective capabilities. For full details, look up the interface `java.lang.reflect.AnnotatedElement` in the reference section.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

`AnnotatedElement` represents a program element that can be queried for annotations. It is implemented by `java.lang.Package`, `java.lang.Class`, and indirectly implemented by the `Method`, `Constructor`, and `Field` classes of `java.lang.reflect`. Annotations on method parameters can be queried with the `getParameterAnnotations( )` method of the `Method` or `Constructor` class.

The following code uses the `isAnnotationPresent( )` method of `AnnotatedElement` to determine whether a method is unstable by checking for an `@Unstable` annotation. It assumes that the `Unstable` annotation type, which we'll define later in the chapter, has runtime retention. Note that this code uses class literals to specify both the class to be checked and the annotation to check for:

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

Class c = WhizzBangClass.class;
Method m = c.getMethod("whizzy", int.class, int.class);
boolean unstable = m.isAnnotationPresent(Unstable.class);
```

`isAnnotationPresent()` is useful for marker annotations. When working with annotations that have members, though, we typically want to know the value of those members. For this, we use the `getAnnotation()` method. And here we see the beauty of the Java annotation system: if the specified annotation exists, the object returned by this method implements the annotation type interface, and you can query the value of any member simply by invoking the annotation type method that defines that member. Consider the `@Reviews` annotation that appeared earlier in the chapter, for example. If the annotation type was declared with runtime retention, you could query it as follows:

```
AnnotatedElement target = WhizzBangClass.class; // the type to query
// Ask for the @Reviews annotation as an object that implements Reviews
Reviews annotation = target.getAnnotation(Reviews.class);
// Reviews has a single member named "value" that is an array of reviews
Review[] reviews = annotation.value();
// Loop through the reviews
for(Review r : reviews) {
    Review.Grade grade = r.grade();
    String reviewer = r.reviewer();
    String comment = r.comment();
    System.out.printf("%s assigned a grade of %s and comment '%s'%n",
                      reviewer, grade, comment);
}
```

Note that these reflective methods correctly resolve default annotation values for you. If an annotation does not include a value for a member with a default value, the default value is looked up within the annotation type itself.

4.3.5. Defining Annotation Types

An annotation type is an interface, but it is not a normal one. An annotation type differs from a normal interface in the following ways:

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

- An annotation type is defined with the keyword `@interface` rather than with `interface`. An `@interface` declaration implicitly extends the interface `java.lang.annotation.Annotation` and may not have an explicit `extends` clause of its own.
- The methods of an annotation type must be declared with no arguments and may not throw exceptions. These methods define annotation members: the method name becomes the member name, and the method return type becomes the member type.
- The return value of annotation methods may be a primitive type, a `String`, a `Class`, an enumerated type, another annotation type, or a single-dimensional array of one of those types.
- Any method of an annotation type may be followed by the keyword `default` and a value compatible with the return type of the method. This strange new syntax specifies the default value of the annotation member that corresponds to the method. The syntax for default values is the same as the syntax used to specify member values when writing an annotation. `null` is never a legal default value.
- Annotation types and their methods may not have type parameters—annotation types and members cannot be made generic. The only valid use of generics in annotation types is for methods whose return type is `Class`. These methods may use a bounded wildcard to specify a constraint on the returned class.

In other ways, annotation types declared with `@interface` are just like regular interfaces. They may include constant definitions and static member types such as enumerated type definitions. Annotation types may also be implemented or extended just as normal interfaces are. (The classes and interfaces that result from doing this are not themselves annotation types, however: annotation types can be created only with an `@interface` declaration.)

We now define the annotation types used in our examples. These examples illustrate the syntax of annotation type declarations and demonstrate many of the differences between `@interface` and `interface`. We start with the simple marker annotation type `Unstable`. Because we used this type earlier in the chapter in a reflection example, its definition includes a meta-annotation that gives it runtime retention and makes it accessible to the reflection API. Meta-annotations are covered below.

```
package com.davidflanagan.annotations;
import java.lang.annotation.*;

/**
 * Specifies that the annotated element is unstable and its API is
 * subject to change.
 */
@Retention(RetentionPolicy.RUNTIME)
public @interface Unstable {}
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

The next annotation type defines a single member. By naming the member `value`, we enable a syntactic shortcut for anyone using the annotation:

```
/**
 * Specifies the author of a program element.
 */
public @interface Author {
    /** Return the name of the author */
    String value();
}
```

The next example is more complex. The `Reviews` annotation type has a single member, but the type of the member is complex: it is an array of `Review` annotations. The `Review` annotation type has three members, one of which has an enumerated type defined as a member of the `Review` type itself, and another of which has a default value. Because the `Reviews` annotation type is used in a reflection example, we've given it runtime retention with a meta-annotation:

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

/**
 * An annotation of this type specifies the results of one or more
 * code reviews for the annotated element
 */
@Retention(RetentionPolicy.RUNTIME)
public @interface Reviews {
    Review[] value();
}

/**
 * An annotation of this type represents a single code review of the
 * annotated element.  Every review must specify the name of the reviewer
 * and the grade assigned to the code.  Optionally, reviews may also include
 * a comment string.
 */
public @interface Review {
    // Nested enumerated type
    public static enum Grade { EXCELLENT, SATISFACTORY, UNSATISFACTORY };

    // These methods define the annotation members
    Grade grade();               // member named "grade" with type Grade
    String reviewer();
    String comment() default "";  // Note default value here.
}
```

Finally, suppose we wanted to annotate methods to list the unchecked exceptions (but not errors) that they might throw. Our annotation type would have a single member of array type. Each element of the array would be the `Class` of an exception. In order to enforce the requirement that only unchecked exceptions are used, we use a bounded wildcard on `Class`:

```
public @interface UncheckedExceptions {
    Class<? extends RuntimeException>[] value();
}
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

4.3.6. Meta-Annotations

Annotation types can themselves be annotated. Java 5.0 defines four standard *meta-annotation* types that provide information about the use and meaning of other annotation types. These types and their supporting classes are in the `java.lang.annotation` package, and you can find complete details in the quick-reference section of the book.

4.3.6.1. Target

The `Target` meta-annotation type specifies the "targets" for an annotation type. That is, it specifies which program elements may have annotations of that type. If an annotation type does not have a `Target` meta-annotation, it can be used with any of the program elements described earlier. Some annotation types, however, make sense only when applied to certain program elements. `Override` is one example: it is only meaningful when applied to a method. An `@Target` meta-annotation applied to the declaration of the `Override` type makes this explicit and allows the compiler to reject an `@Override` when it appears in an inappropriate context.

The `Target` meta-annotation type has a single member named `value`. The type of this member is `java.lang.annotation.ElementType[]`. `ElementType` is an enumerated type whose enumerated values represent program elements that can be annotated.

4.3.6.2. Retention

We discussed annotation *retention* earlier in the chapter. It specifies whether an annotation is discarded by the compiler or retained in the class file, and, if it is retained in the class file, whether it is read by the VM when the class file is loaded. By default, annotations are stored in the class file but not available for runtime reflective access. The three possible retention values (source, class, and runtime) are described by the enumerated type `java.lang.annotation.RetentionPolicy`.

The `Retention` meta-annotation type has a single member named `value` whose type is `RetentionPolicy`.

4.3.6.3. Documented

`Documented` is a meta-annotation type used to specify that annotations of some other type should be considered part of the public API of the annotated program element and should therefore be documented by tools like *javadoc*. `Documented` is a marker annotation: it has no members.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

4.3.6.4. Inherited

The `@Inherited` meta-annotation is a marker annotation that specifies that the annotated type is an inherited one. That is, if an annotation type `@Inherited` is used to annotate a class, the annotation applies to subclasses of that class as well.

Note that `@Inherited` annotation types are inherited only by subclasses of an annotated class. Classes do not inherit annotations from interfaces they implement, and methods do not inherit annotations from methods they override.

The Reflection API enforces the inheritance if the `@Inherited` annotation type is also annotated `@Retention(RetentionPolicy.RUNTIME)`. If you use `java.lang.reflect` to query a class for an annotation of an `@Inherited` type, the reflection code checks the specified class and each of its ancestors until an annotation of the specified type is found or the top of the class hierarchy is reached.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01