

Java 1.5 Tiger

*A Developer's
Notebook™*

Brett McLaughlin
David Flanagan

O'REILLY®

Enumerated Types

In Java 1.4 and below, there were two basic ways to define new types: through classes and interfaces. For most object-oriented programming, this would seem to be enough. The problem is that there are still some very specific cases where neither of these is sufficient, most commonly when you need to define a finite set of allowed values for a specific data type. For instance, you might want a type called `Grade` that can only be assigned values of `A`, `B`, `C`, `D`, `F`, or `Incomplete`. Any other values are illegal for this type. This sort of construct is possible prior to Tiger, but it takes a lot of work, and there are still some significant problems.

Since we're good developers and try our best to avoid a lot of work whenever possible, Sun finally helped us out with the new enumerated type (generally referred to simply as an *enum*). This chapter deals with enums: how to create, use, and program with them.

Creating an Enum

Creating an enumerated type involves three basic components, at a minimum:

- The `enum` keyword
- A name for the new type
- A list of allowed values for the type

There are several optional components that may be defined as well:

- An interface or set of interfaces that the enum implements
- Variable definitions
- Method definitions
- Value-specific class bodies

In this chapter:

- *Creating an Enum*
- *Declaring Enums Inline*
- *Iterating Over Enums*
- *Switching on Enums*
- *Maps of Enums*
- *Sets of Enums*
- *Adding Methods to an Enum*
- *Implementing Interfaces with Enums*
- *Value-Specific Class Bodies*
- *Manually Defining an Enum*
- *Extending an Enum*

These optional components are detailed in the labs throughout this chapter; this lab covers the most basic concepts of enumerated types.

Enums allow you to dump most of your "public static final" variable declarations.

How do I do that?

Example 3-1 is about as basic of an enum as you'll find, representing a simple Grade object.

Example 3-1. A simple enumerated type

```
package com.oreilly.tiger.ch03;

public enum Grade { A, B, C, D, F, INCOMPLETE };
```

You can then define a class that refers to this enum just as it would to any other Java class or interface, as shown in Example 3-2.

More often than not, you'll only need the basic enum functionality.

Example 3-2. Referring to an enum in another class

```
package com.oreilly.tiger.ch03;

public class Student {

    private String firstName;
    private String lastName;
    private Grade grade;

    public Student(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getLastName() {
        return lastName;
    }

    public String getFullName() {
        return new StringBuffer(firstName)
            .append(" ")
            .append(lastName)
    }
}
```

The convention is to use all capital letters for enumerated type identifiers.

"Grade" is used just like any other Java type.

Example 3-2. Referring to an enum in another class (continued)

```
        .toString());
    }

    public void assignGrade(Grade grade) {
        this.grade = grade;
    }

    public Grade getGrade() {
        return grade;
    }
}
```

Pretty basic, isn't it? The final piece is actually using this code in conjunction with the enum, as shown here:

```
public void testGradeAssignment(PrintStream out) throws IOException {
    Student student1 = new Student("Brett", "McLaughlin");
    Student student2 = new Student("Ben", "Rochester");
    Student student3 = new Student("Dennis", "Erwin");

    student1.assignGrade(Grade.B);
    student2.assignGrade(Grade.INCOMPLETE);
    student3.assignGrade(Grade.A);
}
```

*This code is in the
com.oreilly.
tiger.ch03.
GradeTester
class.*

I realize that you may have expected some complex treatment of enums, but I'm not sure I can make it any harder—enums are a nice, elegant feature of the language, and really don't take much explaining—at least for basic usage.

What just happened?

I know there are many of you out there wondering about what goes on under the hood. Here are the highlights about how you can use enums, and their basic construction:

Enums are classes

As a result, you get type-safety, compile-time checking, and the ability to use them in variable declarations. This beats the proverbial pants off of integer constants (see the "What about..." section in this lab).

Enums extend java.lang.Enum

java.lang.Enum is a new class in Tiger, and is not itself an enumerated type. All enumerated types implicitly extend Enum.

*"Grade" is the
type; A, B, C,
and so forth are
the values for
that type. Enum
terminology is a bit
confusing, so it's
good to keep
these straight.*

Enumerated types aren't integers

Each declared value is an instance of the enum class itself; this ensures type-safety and allows for even more compile-time checking.

Enums have no public constructor

This removes the ability to create additional instances of the enum not defined at compile-time. Only those instances defined by the enum are available.

Enum values are public, static, and final

Values cannot be overwritten, changed, or otherwise messed with in ways that affect your programming logic. The enum itself is effectively final, as it cannot be subclassed (see “Extending an Enum”). In fact, the specification says that you are not allowed to declare an enum as final or abstract, as the compiler will take care of those details.

Enums are not “final” when they have value-specific methods, discussed later in the chapter.

Enum values can be compared with == or equals()

Because enums are effectively final, and there is a distinct set of values, you can use—for comparison. Additionally, enumerated types have a working equals(), for use in collection classes (see “Maps of Enums” and “Sets of Enums” later in this chapter).

Enums implements java.lang.Comparable

As a result, enum values can be compared with compareTo(), and ordering occurs in the same order as values are declared in the enum declaration.

Enums override toString()

The toString() method on an enumerated type returns the name of the value. Grade.INCOMPLETE.toString() returns the String “INCOMPLETE”. However, this method isn't final, and can be overridden if desired.

Enums provide a valueOf() method

The static valueOf() method complements toString(). Grade.valueOf(“INCOMPLETE”) returns Grade.INCOMPLETE.

WARNING

If you change the behavior of toString(), you need to also change the behavior of valueOf(). These two methods should always be mirror images of each other.

Enums define a final instance method named ordinal()

ordinal() returns the integer position of each enumerated value, starting at zero, based on the declaration order in the enum. This isn't a method you should use in your own code, but it's used by other enum-related functionality, so is worth knowing about.

Enums define a values() method

values() allows for iteration over the values of an enum, as detailed later in this chapter in "Iterating Over Enums".

What about...

...doing this in Java 1.4 (and previous releases)? At first glance, you may not see all the advantages that enums offer, especially if you're comfortable with static and final variables (essentially constants) in pre-Tiger JDKs. In fact, the Grade class should look an awful lot like the OldGrade class shown in Example 3-3, which is how you might write Grade in a pre-Tiger environment.

Example 3-3. Writing enums in pre-Tiger JDKs

```
package com.oreilly.tiger.ch03;

public class OldGrade {
    public static final int A = 1;
    public static final int B = 2;
    public static final int C = 3;
    public static final int D = 4;
    public static final int F = 5;
    public static final int INCOMPLETE = 6;
}
```

However, there are a lot of problems that aren't immediately apparent. First, consider that the following line of code is legal if you are using OldGrade:

```
student1.assignGrade(1);
```

If you move to Tiger, though, and declare that assignGrade() only accepts a Grade enum, that same line will result in a compiler error:

```
[javac] code\src\ch03\GradeTester.java:19:
        assignGrade(com.oreilly.tiger.ch03.Grade) in
        com.oreilly.tiger.ch03.Student can not be applied to (int)
[javac] student1.assignGrade(1);
```

Using the OldGrade class, you aren't passing in an object of a specific type; you're just passing in an int, that happens (in your specific program implementation) to be associated with the variable name OldGrade.A. You

Your error may look a little different—I formatted this to be readable on the page of a book.

should see the enormous ability to misuse these integer constants, because they're not strongly typed to a specific grade.

Even worse, consider using this same system (Student and OldGrade), but with another "constant" class:

```
public class OldClass {
    public static final int EnglishLit      = 1;
    public static final int Calculus       = 2;
    public static final int MusicTheory    = 3;
    public static final int MusicPerformance = 4;
}
```

Now things get even hairier, because suddenly the following code is legal:

```
student1.assignGrade(OldClass.EnglishList);
```

Some junior programmer's typo suddenly gave this student an A! Enumerated types may seem like a minor convenience, but they turn out to be a major step forward for Java. Use them often, and let the compiler catch your mistakes, rather than a late-night debugging session.

Declaring Enums Inline

While it's useful to create a separate enum class, defined in its own source file, sometimes it's also useful to just define an enum, use it, and throw it away. This is possible through member types.

How do I do that?

Just define the enum within your class, as you would any other member variable. You might need a DownloadStatus enum, for example, but only within a Downloader class:

```
public class Downloader {

    public enum DownloadStatus { INITIALIZING, IN_PROGRESS, COMPLETE };

    // Class body
}
```

Oddly enough, this same code may be written as follows:

```
public class Downloader {

    public static enum DownloadStatus { INITIALIZING, IN_PROGRESS, COMPLETE };

    // Class body
}
```

In this case, the `static` modifier has been added. This has no effective change on the enum, as nested enums are implicitly `static`. In other words, it's sort of like declaring an interface `abstract`—it's redundant. Because of this redundancy, I'd recommend *against* using the `static` keyword in these declarations.

Iterating Over Enums

Ever been given a class with lousy documentation, no source code, and little instruction on its use? Welcome to the loosely knit organization of real-world programmers. In many cases, you can resort to reflection to figure out what a class has to offer in lieu of source code, and of course JavaDoc is always helpful. In the case of enumerated types, though, there's a nice built-in feature: the `values()` method. This method provides access to all of the types within an enum.

Examples in the enum specification also omit "static" in nested declarations.

How do I do that?

Invoking the `values()` method on an enum returns an array of all the values in the type:

```
public void listGradeValues(PrintStream out) throws IOException {
    Grade[] gradeValues = Grade.values();
    for (Grade g : Grade.values()) {
        out.println("Allowed value: " + g + "");
    }
}
```

This is a nice way to get a quick dump of all the allowed values for a particular enum:

```
run-ch03:
[echo] Running Chapter 3 examples from Java 1.5: A Developer's Notebook

[echo] Running GradeTester...
[java] Allowed value: 'A'
[java] Allowed value: 'B'
[java] Allowed value: 'C'
[java] Allowed value: 'D'
[java] Allowed value: 'F'
[java] Allowed value: 'INCOMPLETE'
```

Run this sample with the Ant target "run-ch03".

What just happened?

First, note that type-safety is employed. `values()` doesn't return an array of `String` values—instead it returns an array of `Grade` instances. In the `out.println()` statement, each `Grade` has its `toString()` method

executed, which in turn *does* provide a `String` name for the value. At no point are you working with integer constants or even `String` values—the `Grade` object hides all these implementation details from you, and allows strict compile-time checking.

What about...

...using a `for/in` loop? Well, you're ahead of me—`for/in` isn't covered until Chapter 7. Still, for those of you who are curious, you can indeed perform the same iteration with Tiger's new `for/in` capabilities:

```
// for/in loop
for (Grade g : grade.values()) {
    out.println("Allowed value: '" + g + "'");
}
```

Switching on Enums

As you begin to integrate enums into your own programs, one of the first tasks you'll want to accomplish is using an enum with a `switch` statement. This is a pretty obvious application; there's little value in using enums if you can't easily react to the set of values available.

How do I do that?

Prior to Java 1.4, `switch` only worked with `int`, `short`, `char`, and `byte` values. However, since enums have a finite set of values, Tiger adds `switch` support for them. Here's an example of using an enum in a `switch` statement:

This code assumes that student1 has already been created; this is taken care of in the test class, "GradeTester".

```
public void testSwitchStatement(PrintStream out) throws IOException {
    StringBuffer outputText = new StringBuffer(student1.getFullName());

    switch (student1.getGrade()) {
        case A:
            outputText.append(" excelled with a grade of A");
            break;
        case B: // fall through to C
        case C:
            outputText.append(" passed with a grade of ")
                .append(student1.getGrade().toString());

            break;
        case D: // fall through to F
        case F:
            outputText.append(" failed with a grade of ")
                .append(student1.getGrade().toString());

            break;
        case INCOMPLETE:
    }
```

```

        outputText.append(" did not complete the class.");
        break;
    }

    out.println(outputText.toString());
}

```

The argument to `switch` must be an enumerated value; in this case, the return type of `getGrade()` is `Grade`, which meets these requirements. However, there is another requirement that makes this code a little odd—did you catch it? Note the format of each case clause:

```

case A:
case B:
case C:
case D:
case F:
case INCOMPLETE:

```

See anything missing? How about the enum class identifier:

```

case Grade.A:
case Grade.B:
case Grade.C:
case Grade.D:
case Grade.F:
case Grade.INCOMPLETE:

```

For those of you up on Tiger, this may make you think about the `import static` feature of the language, which I cover in Chapter 8. However, the two have no relation (except perhaps on an implementation level)—Tiger simply *requires* that you not preface each enumerated type with the enum class name. In fact, it's a compilation error if you do! Sort of a nice convenience function, I think.

There's another issue you should be careful about—not handling every enumerated type. In the following version of the `switch`, I've left out handling of `Grade.D`:

```

switch (student1.getGrade()) {
    case A:
        outputText.append(" excelled with a grade of A");
        break;
    case B: // fall through to C
    case C:
        outputText.append(" passed with a grade of ")
            .append(student1.getGrade().toString());
        break;
    case F:
        outputText.append(" failed with a grade of ")
            .append(student1.getGrade().toString());
        break;
    case INCOMPLETE:

```

```
        outputText.append(" did not complete the class.");
        break;
    }
```

It's not completely clear as to if the compiler will be *required* to issue a warning if all types aren't handled; however, it is clear that this is bad coding. You need to be sure that every possible enumerated type is handled, or get ready for some late-night debugging sessions.

You can compile the book's code with warnings turned on with "ant check-compile". All "-Xlint" warnings will be displayed.

TIP

As of this writing, the Tiger compiler did *not* issue a -Xlint warning in this situation.

What just happened?

The handling of an enumerated type by the compiler is a little different than the handling of an integral type. That difference stems from enum values not being compile-time constants; in other words, your code is not turned into the following at compile-time:

```
switch (student1.getGrade()) {
    case 0:
        outputText.append(" excelled with a grade of A");
        break;
    case 1: // fall through to C
    case 2:
        outputText.append(" passed with a grade of ")
            .append(student1.getGrade().toString());
        break;
    case 3: // fall through to F
    case 4:
        outputText.append(" failed with a grade of ")
            .append(student1.getGrade().toString());
        break;
    case 5:
        outputText.append(" did not complete the class.");
        break;
}
```

Instead, assuming that the enum and switch statement exist in the same compilation unit, a *jump table* is created, relating each enumerated type to the value of `ordinal()`, invoked on each type. That results in nearly the same performance as the inlining shown above; the ordinal values aren't inserted into the code, but the compiler can look them up in the jump table extremely quickly. If the enum is changed and recompiled, the jump table is updated, and there's no problem.

More often than not, though, the switch and enum are *not* in the same compilation unit, and this is not possible. In these cases, most compilers turn the switch statement into a series of if/else statements:

```
Grade tmp = student1.getGrade();
if (tmp == Grade.A)
    outputText.append(" excelled with a grade of A");
else if ((tmp == Grade.B) || (tmp == Grade.C))
    outputText.append(" passed with a grade of ")
        .append(student1.getGrade().toString());
else if ((tmp == Grade.D) || (tmp == Grade.F))
    outputText.append(" failed with a grade of ")
        .append(student1.getGrade().toString());
else if (tmp == Grade.INCOMPLETE)
    outputText.append(" did not complete the class.");
```

This isn't efficient as a jump table, but this ensures that if the enum is changed in one compilation unit, the switch statement (in a different unit) continues to function properly. It also removes the need to worry about reordering of an enum, which would affect a jump table.

"default" is best used in this way, to catch unexpected values—it's generally good programming practice to specifically handle every known value, though, as that makes your code much clearer. "default" should be for handling unknown conditions, and not used as a catch-all.

What about...

...using the default keyword? It's perfectly legal, and in fact strongly recommended. Since enum is a new type in Java, it would be easy for someone to come along and add a new type to your enum without you knowing about it:

```
public enum Grade { A, B, C, D, F, INCOMPLETE,
    WITHDREW_PASSING, WITHDREW_FAILING };
```

Now, your switch code will fail because it doesn't account for all the possible Grade values. It's better to plan for this with a little more robust code:

```
switch (student1.getGrade()) {
    case A:
        outputText.append(" excelled with a grade of A");
        break;
    case B: // fall through to C
    case C:
        outputText.append(" passed with a grade of ")
            .append(student1.getGrade().toString());
        break;
    case D: // fall through to F
    case F:
        outputText.append(" failed with a grade of ")
            .append(student1.getGrade().toString());
        break;
    case INCOMPLETE:
        outputText.append(" did not complete the class.");
```

```

        break;
    default:
        outputText.append(" has a grade of ")
            .append(student1.getGrade().toString());
    }

```

An even better idea would be to throw some sort of error on an unexpected type--this will ensure that you and other programmers realize that something is out of sync:

```

    default: throw new AssertionError("Unexpected enumerated value!");

```

Maps of Enums

Once you've gotten your fingers used to typing `public enum`, you'll start to find all sorts of interesting uses for enums. Once you've gotten past the very basic constant-replacement, you'll start to see that they also serve as great keys, or indices, in collection-type structures. Apparently the Sun guys thought the same thing, and provided a nice facility for working with enums as indices.

How do I do that?

In the old, archaic, pre-Tiger days (snicker, snicker), you might have used a constants class like `OldAntStatus` (you'll remember a similar class from Example 3-3):

```

public class OldAntStatus {
    public static final int INITIALIZING = 0;
    public static final int COMPILING   = 1;
    public static final int COPYING     = 2;
    public static final int JARRING     = 3;
    public static final int ZIPPING     = 4;
    public static final int DONE       = 5;
    public static final int ERROR      = 6;
}

```

You might then write a simple array of messages that are associated with each of these status codes:

```

String[] antMessages = new String[] {
    "Initializing Ant...", // INITIALIZING
    "Compiling Java classes...", // COMPILING
    "Copying files...", // COPYING
    "JARring up files...", // JARRING
    "ZIPping up files...", // ZIPPING
    "Build complete.", // DONE
    "Error occurred." // ERROR
}

```

You can then access the appropriate message using the constant:

```
int antStatus = antProcess.getStatus();
out.println("ant> " + antMessages[antStatus]);
```

It's a pretty valid desire to want to accomplish the same sorts of tasks with enumerated types, and get all the benefits of enums along the way. You'll need to use the `java.util.EnumMap` class to accomplish this, which is a new collection type just perfect for the job. First, you need to define the enum you want to use for a keyset, as shown in Example 3-4.

Example 3-4. An enum to use for a keyset

```
package com.oreilly.tiger.ch03;

public enum AntStatus {
    INITIALIZING,
    COMPILING,
    COPYING,
    JARRING,
    ZIPPING,
    DONE,
    ERROR
}
```

You can now create a new `EnumMap`, in conjunction with generics—declare the enumerated type you want to use for the key, and the class type you want to use for the value:

```
EnumMap<AntStatus, String> antMessages;
```

Then, when creating a new instance, you pass the `EnumMap` the `Class` object for the enum used for the keyset:

```
EnumMap<AntStatus, String> antMessages =
    EnumMap<AntStatus, String>(AntStatus.class);
```

Remember to declare the types being used, key and value, in both the variable declaration and the variable instantiation.

TIP

This should further emphasize the fact that enums are really just Java classes that the compiler handles specially. This is discussed in more detail in the “Creating an Enum” lab.

Once the initialization is taken care of, things get very simple. You just seed the values, and use them:

```
public void testEnumMap(PrintStream out) throws IOException {
    // Create a map with the key and a String message
    EnumMap<AntStatus, String> antMessages =
        new EnumMap<AntStatus, String>(AntStatus.class);

    // Initialize the map
    antMessages.put(AntStatus.INITIALIZING, "Initializing Ant...");
```

This code is in `AntStatusTester.java`.

```

antMessages.put(AntStatus.COMPILING, "Compiling Java classes...");
antMessages.put(AntStatus.COPYING, "Copying files...");
antMessages.put(AntStatus.JARRING, "JARRing up files...");
antMessages.put(AntStatus.ZIPPING, "ZIPPING up files...");
antMessages.put(AntStatus.DONE, "Build complete.");
antMessages.put(AntStatus.ERROR, "Error occurred.");

// Iterate and print messages
for (AntStatus status : AntStatus.values() ) {
    out.println("For status " + status + ", message is: " +
        antMessages.get(status));
}
}

```

Running this code nicely prints out all the status codes, and the associated message with each:

```

[echo] Running AntStatusTester...
[java] For status INITIALIZING, message is: Initializing Ant...
[java] For status COMPILING, message is: Compiling Java classes...
[java] For status COPYING, message is: Copying files...
[java] For status JARRING, message is: JARRing up files...
[java] For status ZIPPING, message is: ZIPPING up files...
[java] For status DONE, message is: Build complete.
[java] For status ERROR, message is: Error occurred.

```

Of course, as an added benefit, EnumMap protects you from mis-ordering when initializing values, reordering in the enumerated type, and just about any other strange situation that can arise from more than one person working on code at the same time.

Sets of Enums

Another common usage of constants is to represent a feature set of a particular item, such as a car, or even something really exciting, like a guitar. You could then use bitwise operators to compare or check for specific features. As in the lab on “Maps of Enums,” the Java language folks realized this, and added another collection class for the purpose, `java.util.EnumSet`.

How do I do that?

First, examine the way this scenario might be handled in pre-Tiger days. Example 3-5 is a simple class that defines some common guitar features.

Example 3-5. Guitar feature set

```

package com.oreilly.tiger.ch03;

public class OldGuitarFeatures {

```

Example 3-5. Guitar feature set (continued)

```
public static final int ROSEWOOD      = 0x01; // back/sides
public static final int MAHOGANY     = 0x02; // back/sides
public static final int ZIRICOTE     = 0x04; // back/sides

public static final int SPRUCE       = 0x10; // top
public static final int CEDAR        = 0x12; // top

public static final int AB_ROSETTE   = 0x30; // abalone rosette
public static final int AB_TOP_BORDER = 0x32; // abalone top border

public static final int IL_DIAMONDS  = 0x40; // diamond/square inlay
public static final int IL_DOTS      = 0x42; // dots inlays
}
```

These are all features, and are all represented by powers of two. That allows them to be combined like this:

```
int bourgeoisD150 = OldGuitarFeatures.ROSEWOOD |
                  OldGuitarFeatures.SPRUCE |
                  OldGuitarFeatures.AB_ROSETTE |
                  OldGuitarFeatures.IL_DIAMONDS;
```

With this initial work done (you could represent every guitar in a line this way), you can test a specific guitar for a specific features, using the bitwise AND operator:

```
boolean hasAbRosette = (bourgeoisD150 & OldGuitarFeatures.IL_DIAMONDS) != 0;
```

Looking at the constants in `OldGuitarFeatures`, you should see that they are just another case of an enumerated type, and could be represented in Tiger as shown in Example 3-6.

Example 3-6. Representing guitar features in Tiger

```
package com.oreilly.tiger.ch03;

public enum GuitarFeatures {
    ROSEWOOD, // back/sides
    MAHOGANY, // back/sides
    ZIRICOTE, // back/sides

    SPRUCE, // top
    CEDAR,  // top

    AB_ROSETTE, // abalone rosette
    AB_TOP_BORDER, // abalone top border

    IL_DIAMONDS, // diamond/square inlay
    IL_DOTS      // dots inlays
}
```

Be sure and continue to compile with the "-source 1.5" switch. Using the provided Ant scripts takes care of this, by the way.

However, operating on these constants with bitwise operators isn't possible, at least without the help of a new class, `java.util.EnumSet`. Here are the methods of this class you should be concerned with, most of which are factories:

```
// Returns a new EnumSet with all elements from the supplied type
public static EnumSet allOf(Class elementType);

// Returns a new EnumSet of the same type as the supplied set, but
// with all the values not in the supplied set; a mirror image
public static EnumSet complementOf(EnumSet e);

// Returns a new EnumSet from the provided collection
public static EnumSet copyOf(Collection c);

// Returns a new EnumSet with no values in it
public static EnumSet noneOf(Class elementType);

// Various methods to create an EnumSet with the supplied elements in it
public static EnumSet of(E e1, E e2, E e3, E e4, E e5);

//Varags version
public static EnumSet of(E... e);

// Creates an EnumSet with a range of values
public static EnumSet range(E from, E to);

// returns a copy of the current set - not a factory method
public EnumSet clone();
```

You may have to look at this for a minute to get the sense of things—the format is rather odd. But, to create a new `EnumSet` of all guitar features, just use the following:

```
EnumSet allFeatures = EnumSet.allOf(GuitarFeatures);
```

Simple enough, right? If you want only the back and side woods, you could use this notation:

```
EnumSet backSides = EnumSet.of(GuitarFeatures.ROSEWOOD,
                               GuitarFeatures.MAHOGANY,
                               GuitarFeatures.ZIRICOTE);
```

You could also use the `range()` operator:

```
EnumSet backSides = EnumSet.range(GuitarFeatures.ROSEWOOD,
                                   GuitarFeatures.ZIRICOTE);
```

WARNING

This is really a bad idea—reordering of the enum screws this up, and you can never really completely insulate yourself from that possibility. Always use `of()` instead of `range()`, if at all possible.

`complementOf()` is also a handy method:

```
EnumSet noAbalone = EnumSet.complementOf(  
    EnumSet.of(GuitarFeatures.AB_ROSETTE, GuitarFeatures.AB_TOP_BORDER));
```

With your setup done, you can just use the `contains()` method, available to all collection classes, to check for a value or values:

```
EnumSet bourgeoisD150 = EnumSet.of(GuitarFeatures.ROSEWOOD,  
    GuitarFeatures.SPRUCE,  
    GuitarFeatures.AB_ROSETTE,  
    GuitarFeatures.IL_DIAMONDS);  
  
boolean hasAbRosette = bourgeoisD150.contains(GuitarFeatures.AB_ROSETTE);
```

- You can't use varargs to work with the `of()` method, due to the usage of generic types in the factory method. Variable arguments are detailed in Chapter 5.

Adding Methods to an Enum

Those of you paying attention should have noticed something pretty important that I said earlier in this chapter in the “Creating an Enum” lab—that enums are just Java classes. They have some special behavior that you get for free, but ultimately they are indeed compiled classes. What's important about this is that it implies that you can do some pretty cool things with enums beyond the basics you've seen so far—such as adding methods to the enum. This is a great way to make your enum even more valuable to application programmers.

How do I do that?

Adding methods to an enum works just like adding methods to a normal Java class. Example 3-7 is a beefed-up version of `GuitarFeatures` that adds several new methods.

Example 3-7. Adding methods to an enum

```
package com.oreilly.tiger.ch03;  
  
public enum GuitarFeatures {  
  
    ROSEWOOD(0),          // back/sides  
    MAHOGANY(0),          // back/sides  
    ZIRICOTE(300),        // back/sides  
  
    SPRUCE(0),            // top  
    CEDAR(0),             // top  
  
    AB_ROSETTE(75),       // abalone rosette  
    AB_TOP_BORDER(400),   // abalone top border  
  
    IL_DIAMONDS(150),     // diamond/square inlay
```

Example 3-7. Adding methods to an enum (continued)

```
IL_DOTS(0);           // dots inlays

/** The upcharge for the feature */
private float upcharge;

GuitarFeatures(float upcharge) {
    this.upcharge = upcharge;
}

public float getUpcharge() {
    return upcharge;
}

public String getDescription() {
    switch(this) {
        case ROSEWOOD:      return "Rosewood back and sides";
        case MAHOGANY:      return "Mahogany back and sides";
        case ZIRICOTE:      return "Ziricote back and sides";
        case SPRUCE:        return "Sitka Spruce top";
        case CEDAR:         return "Wester Red Cedar top";
        case AB_ROSETTE:    return "Abalone rosette";
        case AB_TOP_BORDER: return "Abalone top border";
        case IL_DIAMONDS:   return "Diamonds and squares fretboard inlay";
        case IL_DOTS:       return "Small dots fretboard inlay";
        default:            return "Unknown feature";
    }
}
}
```

There are quite a few things here that you'll need to take note of. First, the class now has a constructor that takes in a float parameter for the upcharge of each feature. As a result, each enumerated type now passes in a parameter to the constructor:

```
ROSEWOOD(0),          // back/sides
MAHOGANY(0),          // back/sides
ZIRICOTE(300),        // back/sides

SPRUCE(0),            // top
CEDAR(0),             // top

AB_ROSETTE(75),       // abalone rosette
AB_TOP_BORDER(400),   // abalone top border

IL_DIAMONDS(150),    // diamond/square inlay
IL_DOTS(0);           // dots inlays
```

This looks a little odd in the code, but opens up a world of possibilities for information to be passed in for each value. You should also notice that the final value is followed by a semicolon. This denotes the end of the values section, and is required.

Then, variables are declared, and methods appear, just like any other class. In the example, `getUpcharge()` returns the value supplied to the constructor, and `getDescription()` supplies a human-readable version of the feature. You'll see that `switch` is used, as described in "Switching on Enums," and this makes the method body simple to read and understand.

WARNING

You cannot put your variable declarations before the enumerated values. The following code, for example, won't compile:

```
public enum GuitarFeatures {
    private float upcharge;

    ROSEWOOD,
    MAHOGANY,    // etc.
}
```

All declarations must follow the enumerated type declarations.

What about...

...limiting access to the enum constructor? Enum constructors are implicitly private, so this is taken care of for you. In some programming techniques, such as singletons, access modifiers are placed in front of the constructor so that it can't be directly accessed:

```
public enum GuitarFeatures {

    ROSEWOOD(0),        // back/sides
    MAHOGANY(0),        // back/sides
    ZIRICOTE(300),      // back/sides

    SPRUCE(0),          // top
    CEDAR(0),           // top

    AB_ROSETTE(75),     // abalone rosette
    AB_TOP_BORDER(400), // abalone top border

    IL_DIAMONDS(150),  // diamond/square inlay
    IL_DOTS(0);        // dots inlays
}
```

```

    /** The upcharge for the feature */
    private float upcharge;

    private GuitarFeatures(float upcharge) {
        this.upcharge = upcharge;
    }

    // Other method bodies
}

```

This compiles, but it just explicitly does what the compiler takes care of for you—making the constructor private. However, you *cannot* supply the standard public modifier:

```

public GuitarFeatures(float upcharge) {
    this.upcharge = upcharge;
}

```

If you try this, you’ll get a compiler error:

```

[javac] src\ch03\GuitarFeatures.java:21:
        modifier public not allowed here
[javac]   public GuitarFeatures(float upcharge) {
[javac]           ^

```

So leave *all* modifiers off of enum constructors.

Implementing Interfaces with Enums

Now that you’re starting to add methods to your enums (see the previous lab on “Adding Methods to an Enum” for details if you’re skipping around), you may find that you want to define methods in an interface, and implement that interface with one, two, or even more enums. You may also want to have an enum implement an interface, and have classes implement that same interface. All this is possible, and even quite simple.

How do I do that?

Example 3-8 is a very simple interface that could describe enumerated types that represent features on all sorts of instruments (not just guitars).

Example 3-8. Base interface for feature enums

```

package com.oreilly.tiger.ch03;

public interface Features {

```

Example 3-8. Base interface for feature enums (continued)

```
/** Get the upcharge for this feature */
public float getUpcharge();

/** Get the description for this feature */
public String getDescription();
}
```

It's trivial to make `GuitarFeatures` implement this interface, as the methods are already written:

```
public enum GuitarFeatures implements Features {
```

Now you can create `BanjoFeatures`, `MandolinFeatures`, and more, all using the same interface as a starting point. This creates a nice sense of uniformity among your enums, and is highly recommended.

Value-Specific Class Bodies

In covering the more advanced features of enums, I can't leave out the ability to define *value-specific class bodies*. That sounds sort of fancy, but all it means is that each enumerated value within a type can define value-specific methods. This is a rather obscure bit of functionality, but sort of cool to talk about around the water cooler.

How do I do that?

Example 3-9 is an example of a class that determines how the `perform()` method is executed based on the enumerated value. It's a perfect example of value-specific class bodies.

Example 3-9. Value-specific class bodies in an enum

```
// These are the opcodes that our stack machine can execute.
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 put the result
    ADD(0) {
        public void perform(StackMachine machine, int[] operands) {
```

This example is lifted straight out of Java in a Nutshell, Fifth Edition (O'Reilly).

Example 3-9. Value-specific class bodies in an enum (continued)

```
        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.pop() == 0) machine.setPC(operands[0]);
    }
}; // Remember the required semicolon after last enum value

// 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);
}
```

Skipping past the individual types (which you should already understand), the method that each value should implement is defined: `perform()`, which takes two arguments. Finally, each value is followed by an opening curly brace, one or more value-specific methods, and then a closing curly brace. This works in conjunction with any constructor that must be supplied a value, as this enum has. The end result, frankly, is one of the oddest looking Java constructs you'll ever see.

I suppose you could really clutter things up with generics and varargs, but you get the idea... value-specific class bodies are often a pain to debug for even mid-level programmers, because of their unusual syntax.

What just happened?

In the lab on “Creating an Enum,” I mentioned that enumerated type values are created and marked as `final` (in addition to being `public` and `static`), ensuring that they aren't changed by some malicious or unknowing programmer. In the case of a value-specific class body, though, this isn't possible. Instead, an anonymous subclass of the type is created, and the value becomes a singleton instance of that subclass. This still ensures that multiple instances of the same value aren't floating around, but it does change what's going on at the compiler level a bit. Despite this, you still can't extend an enum (see “Extending an Enum” for more details).

What about...

...just using a more generic method that determines what to do based on a switch statement? Well, that's a better idea, to be honest. Here's the (much cleaner) way to write OpCode:

```
// These are the the opcodes that our stack machine can execute.
abstract static enum Opcode {
    PUSH(1),
    ADD(0),
    BEZ(1); // Remember the required semicolon after last enum value

    int numOperands;

    Opcode(int numOperands) { this.numOperands = numOperands; }

    public void perform(StackMachine machine, int[] operands) {
        switch(this) {
            case PUSH: machine.push(operands[0]); break;
            case ADD: machine.push(machine.pop() + machine.pop()); break;
            case BEZ: if (machine.pop() == 0) machine.setPC(operands[0]); break;
            default: throw new AssertionError();
        }
    }
}
```

This is so painfully simpler than the first version of OpCode that I hesitated to even include this lab—but for completeness, here it is. If at all possible, though, consider using switch in your method bodies to direct program flow, rather than value-specific class bodies.

Manually Defining an Enum

You'll recall from the first lab, "Creating an Enum," that all enums implicitly extend the new `java.lang.Enum` class. This class looks a bit like Example 3-10; I've trimmed the method implementations and just left the declarations in for clarity.

Example 3-10. The `java.lang.Enum` class

```
package java.lang;

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

    protected Enum(String name, int ordinal);

    protected Object clone();
    public int compareTo(E o);
    public boolean equals(Object other);
```

This code listing is extracted from the Java 1.5 JavaDoc. Source code isn't available as of the time of this writing.

Example 3-10. The `java.lang.Enum` class (continued)

```
public Class<E> getDeclaringClass();
public int hashCode();
public String name();
public int ordinal();
public String toString();
public static <T extends Enum<T>> T valueOf(Class<T> enumType, String name);
}
```

If you're a bit of a hack, that may get your mind wandering...couldn't I just manually define my own enum, then? Good question.

How do I do that?

You don't—at least, not in Tiger. While this is very much an accessible class, and is indeed the base class of all enumerated types in Tiger, the compiler won't let you extend it, as Example 3-11 tries to do.

Example 3-11. Attempting to extend `java.lang.Enum`

```
package com.oreilly.tiger.ch03;

public class ExtendedEnum extends Enum {
}
```

Attempting to compile this class give you the following error:

```
[javac] src\ch03\ExtendedEnum.java:3:
         classes cannot directly extend java.lang.Enum
[javac] public class ExtendedEnum extends Enum {
[javac]         ^
```

Extending an Enum

It's often easy to define a hierarchy of enumerations. In this scenario, one enum represents a base type of allowed values. Subclasses of that enum would add additional values to the base enum, perhaps specialized to a certain task.

How do I do that?

Here's another one of those pesky, "You don't" labs. Tiger does not allow extension of an enum. For example, consider Example 3-12, a simple extension of the `Grade` enum defined back in Example 3-1.

Example 3-12. Extending the Grade enum

```
package com.oreilly.tiger.ch03;  
  
public enum CollegeGrade extends Grade { DROP_PASSING, DROP_FAILING }
```

In theory, this would take the values `Grade.A`, `Grade.B`, and so forth, and add to them two new values, `CollegeGrade.DROP_PASSING` and `CollegeGrade.DROP_FAILING`. However, you'll get compilation errors if you try this:

```
compile-ch03-errors:  
  [echo] Compiling all Java files...  
  [javac] Compiling 13 source files to classes  
  [javac] src\ch03\CollegeGrade.java:3: '{' expected  
  [javac] public enum CollegeGrade extends Grade {DROP_PASSING, DROP_  
FAILING}  
  [javac]                                     ^  
  [javac] src\ch03\CollegeGrade.java:3: <identifier> expected  
  [javac] public enum CollegeGrade extends Grade {DROP_PASSING, DROP_  
FAILING}  
  [javac]                                     ^  
  [javac] 2 errors
```

Use the Ant target "compile-ch03-errors" to try and compile the `CollegeGrade`. java source file.

If you find a sneaky way to extend an enum, let us know! We'll add it to the next edition.

What about...

...using a class to extend an enum, instead of another enum? That doesn't work either. There's just no getting around this limitation, at least that I've been able to find..