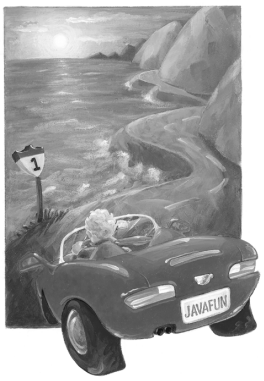
Chapter 6



Static, Final, and Enumerated Types

- What Field Modifier static Means
- What Field Modifier final Means
- Why Enumerate a Type?
- Statements Updated for Enumerations
- More Complicated Enumerated Types
- Some Light Relief—The Haunted Zen Garden of Apple

New!

Enumerated types were brought into Java with the JDK 1.5 release. They are not a new idea in programming, and lots of other languages already have them. The word "enumerate" means "to specify individually". An enumerated type is one where we specify individually (as words) all the legal values for that type.

For a type that represents t-shirt sizes, the values might be *small, medium, large, extraLarge*. For a bread flavors type, some values could be *wholewheat, ninegrain, rye, french, sourdough*. A DaysOfTheWeek enumerated type will have legal values of *Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday.*

The values have to be identifiers. In the USA, ladies' dress sizes are 2, 4, 6, 8, 10, 12, 14, 16. As a Java enumeration, that would have to be represented in words as two, four, or any other characters that form an identifier, such as size2, size4 etc.

When you declare a variable that belongs to an enumerated type, it can only hold one value at a time, and it can't hold values from some other type. A t-shirt size enum variable can't hold "large" and "small" simultaneously, just as an int can't

hold two values simultaneously. You can't assign "Monday" to a t-shirt size variable. Though enumerated types aren't essential, they make some kinds of code more readable.

---

enum is a new keyword

Although JDK 1.5 introduced extensive language changes, "enum" is the only new keyword brought into the language. If any of your existing programs use the word "enum" as an identifier, you will have to change them before you can use JDK.5 features.

The identifier enum might well be in programs that use the older class java.util.Enumeration. That class has nothing to do with the enum type, but is a way of iterating through all the objects in a data structure class. Many people (including me) declared variables such as

```
java.util.Enumeration  enum;
```

The java.util.Enumeration class has been obsoleted by a class called Iterator, also in the java.util package, so if you're updating some code to change a variable called "enum", you might want to modify it to use an iterator too. We cover iterators in Chapter 16.

Before JDK 1.5, a common way to represent enumerations was with integer constants, like this:

```
class Bread {
    static final int wholewheat = 0;
    static final int ninegrain = 1;
    static final int rye = 2;
    static final int french = 3;
}
```

then later

```
int todaysLoaf = rye;
```

In the new enum scheme, enumerations are references to one of a fixed set of objects that represent the various possible values. Each object representing one of the choices knows where it fits in the order, its name, and optionally other information as well. Because enum types are implemented as classes, you can add your own methods to them!

The main purpose of this chapter is to describe enumerated types in detail. To do that, we first need to explain what the *field modifiers* "static" and "final" mean. Here's the story in brief:

- The keyword `final` makes the declaration a constant.

- The keyword `static` makes the declaration belong to the class as a whole. A static field is shared by all instances of the class, instead of each instance having its own version of the field. A static method does not have a "this" object. A static method can operate on someone else's objects, but not via an implicit or explicit *this*.

  The method where execution starts, `main()`, is a static method. The purpose of `main()` is to be an entry point to your code, not to track the state of one individual object. Static "per-class" declarations are different from all the "per-object" data you have seen to date.

The values in enumerated types are always implicitly static and final. The next two sections, *What Field Modifier static Means* and *What Field Modifier final Means*, have a longer explanation of the practical effect of these field modifiers. After that, we'll get into enumerated types themselves.

What Field Modifier `static` Means

We have seen how a class defines the fields and methods that are in an object, and how each object has its own storage for these members. That is usually what you want.

Sometimes, however, there are fields of which you want only one copy, no matter how many instances of the class exist. A good example is a field that represents a total. The objects contain the individual amounts, and you want a single field that represents the total over all the existing objects of that class. There is an obvious place to put this kind of "one-per-class" field too—in a single object that represents the class. Static fields are sometimes called "class variables" because of this.

You could put a total field in every object, but when the total changes you would need to update every object. By making total a *static* field, any object that wants to reference total knows it isn't instance data. Instead it goes to the class and accesses the single copy there. There aren't multiple copies of a static field, so you can't get multiple inconsistent totals.

Static is a really poor name

Of all the many poorly chosen names in Java, "static" is the worst. The keyword is carried over from the C language, where it was applied to storage which can be allocated statically (at compile time). Whenever you see "static" in Java, think "once-only" or "one-per-class."

What you can make `static`

You can apply the modifier `static` to four things in Java:

- **Data.** This is a field that belongs to the class, not a field that is stored in each individual object.
- **Methods.** These are methods that belong to the class, not individual objects.
- **Blocks.** These are blocks within a class that are executed only once, usually for some initialization. They are like instance initializers, but execute once per class, not once per object.
- **Classes.** These are classes that are nested in another class. Static classes were introduced with JDK 1.1.

We'll describe static data and static methods in this chapter. Static blocks and static classes are dealt with later on.

Static data

Static data belongs to the class, not an individual object of the class. There is exactly one instance of static data, regardless of how many objects of the class there are. To make a field "per-class," apply the keyword "static," as shown here.

```
class Employee {
     int    id;                    // per-object field
     int    salary;                // per-object field

     static int total; // per-class field (one only)

            ...
}
```

Every Employee object will have the employee_id and salary fields. There will be one field called totalPayroll stored elsewhere in an object representing the Employee class.

Because static data is *declared* in the class right next to the instance data, it's all too easy to overlook that static data is *not kept* in each object with its instance data. Make sure you understand this crucial point before reading on. Figure 6–1 represents the previous code in the form of a diagram.

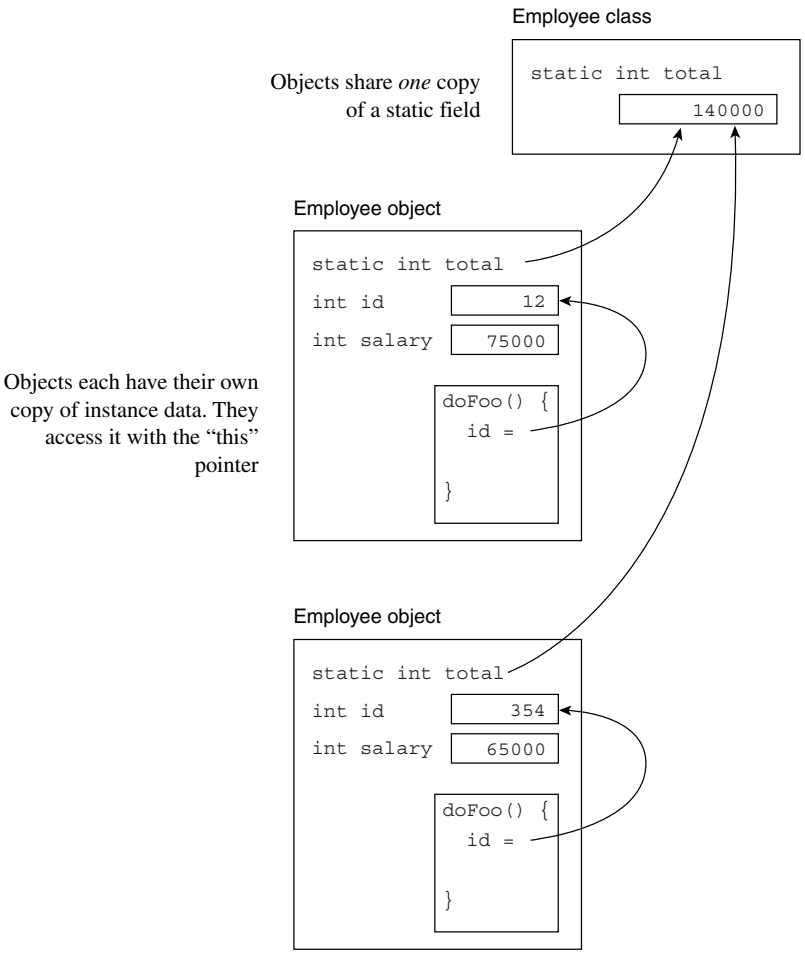


Figure 6–1 There is one copy of a Static field, shared by each object

In methods inside the class, static data is accessed by giving its name just like instance data.

```
salary = 90000;
total = this.total + this.salary;
```

It's legal but highly misleading to qualify the name of a static field with "this." The "this" variable points to an instance, but static data doesn't live in an instance. The compiler knows where the static data really is, and generates code to access the field in the class object.

Outside the class, static data can be accessed by prefixing it with the name of the class *or* the name of an object reference. It is considered poor form to use the object reference method. It confuses the reader into mistaking your static member for an instance member.

```
Employee newhire = new Employee();

// static reference through the class (preferred)
Employee.total += 100000;
```

Static methods

Just as there can be static data that belongs to the class as a whole, there can also be static methods, also called *class methods*. A class method does some class-wide operations and is not applied to an individual object. Again, these are indicated by using the `static` modifier before the method name.

The `main()` method where execution starts is static.

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

If main weren't static, if it were an instance method, some magic would be needed to create an instance before calling it, as is done for applets and servlets.

Any method that doesn't use instance data is a candidate to be a static method. The conversion routines in the wrappers for the primitive types are static methods. If you look at the source code for java.lang.Integer, you'll see a routine like this

```
public static int parseInt(String s)
                                throws NumberFormatException {
     // statements go here.
}
```

The method is a utility that reads the String passed to it as an argument, and tries to turn it into an int return value. It doesn't do anything with data from a specific Integer object (there isn't even an Integer object involved in the call). So `parseInt()` is properly declared as static. It wouldn't be actively harmful to make it an instance method, but you would then need to whisk up an otherwise unnecessary Integer object on which to invoke it. Here's an example of calling the static method parseInt:

```
int i = Integer.parseInt("-2048");
```

The Java Language Specification says "A class method is always invoked without reference to a particular object" (section 8.4.3.2). So some compilers generate an error if you invoke a static method through an instance variable. Other compilers take the view "it's OK to reach static data through an instance reference (and the JLS has an example of this), so it should be OK for static methods too". Stick to

invoking static methods using the class name, to avoid compiler problems and to show other programmers that this a class method.

A common pitfall with static methods

A common pitfall is to reference *per-object* members from a *static* method. This "does not compute". A static method isn't invoked on an object and doesn't have the implicit "this" pointer to individual object data, so the compiler won't know which object you want. You'll get an error message saying "Can't make static reference to non-static variable."

Java novices often make this mistake when they write their first class with several methods. They know the `main()` method has to be static, but they try to invoke the instance methods from inside `main`. The simplest workaround is to declare an instance of the class in question inside `main()`, and invoke the methods on that.

```
class Timestamp {
    void someMethod() { // ...

    public static void main(String[] args) {
        someMethod(); // NO! does not work

        Timestamp ts = new Timestamp();
        ts.someMethod(); // Yes! does work
```

Another workaround is to add the `static` modifier to everything you reference. Only use this kludge for small test programs.

What Field Modifier `final` Means

This section looks at `final`, which makes something constant. Why was the word "const" or "constant" not chosen? Because "final" can also be applied to methods, as well as data, and the term "final" makes better sense for both.

A class or a class member (that is, a data field or a method) can be declared final, meaning that once it is given a value it won't change. We will look at what it means for a class or a method not to change in Chapter 8. A couple of final data declarations are:

```
final static int myChecksum = calculateIt();
final Timestamp noon = new Timestamp(12, 00, 00);
final int universalAnswer = 42;
```

When a reference variable is declared final, it means that you cannot change that variable to point at some other object. You can, however, access the variable and change its fields through that final reference variable. The reference is final, not the referenced object.

JDK 1.1 introduced the ability to mark method arguments and variables local to a method as final, such as:

```
void someMethod(final MyClass c, final int a[]) {
     c.field = 7;       // allowed
     a[0] = 7;          // allowed
     c = new MyClass(); // final means this line is NOT allowed
     a = new int[13];   // final means this line is NOT allowed
}
```

Programmers rarely use this, because it clutters up the signature and makes the parameter names harder to read. That's a pity. Marking a declaration as final is a clue to the compiler that certain optimizations can be made. In the case of final primitive data, the compiler can often substitute the value in each place the name is used, in an optimization known as *constant propagation*. As my friend, talented compiler-writer and builder of battle robots Brian Scearce pointed out, that in turn may lead to other optimizations becoming possible.

The "blank final variable"

JDK 1.1 also introduced something called a *blank final variable,* which is simply a final variable (of any kind) that doesn't have an initializer. A blank final variable must be assigned an initial value, and that can be assigned only once. If you give a value to a blank final in a constructor, every constructor must give it a value. This is because you don't know which constructor will be called, and it must end up with an initialization.

```
class Employee {

     final String name;// blank final variable - has no initializer

     Employee (String s) {  // constructor
          name = s;         // the blank final is initialized
     }

    // more stuff
}
```

Use a blank final when you have a value that is too complicated to calculate in a declaration, or that might cause an exception condition (more on that later), or where the final value depends on an argument to a constructor.

That completes the description of static and final. The next section, *Why Enumerate a Type?*, explains a new feature in Java, the enumerated type, which is built up from final static values.

Why Enumerate a Type?

Here's the older approach of simulating enumerations:

```
class Bread {
    static final int wholewheat = 0;
    static final int ninegrain = 1;
    static final int rye = 2;
    static final int french = 3;
}
```

You would then declare an int variable and let it hold values from the Bread class, e.g.

```
    int todaysLoaf = Bread.rye;
```

Drawbacks of using ints to enumerate

Using final ints to represent values in an enumeration has at least three drawbacks.

- All the compiler tools (debugger, linker, run-time, etc.) still regard the variables as ints. They are ints. If you ask for the value of todaysLoaf, it will be 2, not "rye". The programmer has to do the mapping back and forth mentally.
- The variables aren't typesafe. There's nothing to stop `todaysLoaf` getting assigned a value of 99 that doesn't correspond to any Bread value. What happens next depends on how well the rest of your code is written, but the best case is that some routine notices pretty quickly and throws an exception. The worst case is that your computer-controlled bakery tries to bake "type 99" bread causing an expensive sticky mess.
- Use of integer constants makes code "brittle" (easily subject to breakage). The constants get compiled into every class that use them. If you update the class where the constants are defined, you must go and find all the users of that class, and recompile them against the new definitions. If you miss one, the code will run but be subject to subtle bugs.

How enums solve these issues

Enumerated types were introduced with JDK 1.5 to address these limitations. Variables of enumerated types

- Are displayed to the programmer or user as Strings, not numbers
- Can only hold values defined in the type
- Do not require clients to be recompiled when the enumeration changes

Enumerated types are written using a similar syntax to class declarations, and you should think of them as being a specialized sort of class. An enumerated type

definition can go in a file of its own, or in a file with other classes. A public enumeration must be in a file of its own with the name matching the enumeration name.

You might create an enumerated type to represent some bread flavors. It would be defined like this:

```
enum Bread { wholewheat, ninegrain, rye, french }
```

That lets you declare variables of type Bread in the usual way:

```
Bread todaysLoaf;
```

You can assign an enum value to a variable of Bread type like this:

```
todaysLoaf = Bread.rye;
```

All the language tools know about the symbolic names. If you print out a Bread variable, you get the string value, not whatever numeric constant underlies it internally.

```
System.out.println("bread choice is: " + todaysLoaf);
```

This results in output of:

```
bread choice is: rye
```

How enums are implemented

Under the covers, enum constants are static final objects declared (by the compiler) in their enum class. An enum class can have constructors, methods, and data. Enum variables are merely pointers to one of the static final enum constant objects.

You'll understand enums better if you know that the compiler treats them approximately the same way as if it had seen this source code:

```
class Bread extends Enum {
   // constructor
   public Bread(String name, int position) { super(name, position); }

   public static final Bread wholewheat = new Bread("wholewheat",0);
   public static final Bread ninegrain = new Bread("ninegrain", 1);
   public static final Bread rye = new Bread("rye", 2);
   public static final Bread french = new Bread("french", 3);

   // more stuff here
}
```

This is an approximation because the parent class, java.lang.Enum, uses a generic parameter, and we cover generics later. Bringing generics into the definition of Enum was an unnecessary complication aimed at improving the type-safety of enums. The work could and should have been moved into the compiler. The previous code should give you a good idea of how enums are represented.

Namespaces in Java and in enumerations

Namespace isn't a term that occurs in the Java Language Specification. Instead, it's a compiler term meaning "place where a group of names are organized as a whole." Some older languages only have one global namespace that holds the names of all methods and variables. Along with each name, the compiler stores information about what type it is, and other details.

Java has many namespaces. All the members in a class form a namespace. All the variables in a method form a namespace. A package forms a namespace. Even a local block inside a method forms a namespace.

A compiler will look for an identifier in the namespace that most closely encloses it. If not found, it will look in successively wider namespaces until it finds the first occurrence of the correct identifier. You won't confuse Java if you give the same name to a method, to a data field, and to a label. It puts them in different namespaces. When the compiler is looking for a method name, it doesn't bother looking in the field namespace.

Each enumeration has its own namespace, so it is perfectly valid for enumeration values to overlap with other enums or other variables, like this:

```
enum Fruit { peach, orange, grapefruit, durian }
enum WarmColor { peach, red, orange }
```

Some more Java terminology: the enumeration values apple, red, peach, plum and orange are known as *enum constants*. The enumeration types Fruit and WarmColor are *enum types.*

Enum constants make software more reliable

Here's an amazing thing: the constants that represent the enumeration values are not compiled into other classes that use the enumeration. Instead, each enum constant (like Bread.rye previously) is left as a symbolic reference that will be linked in at run-time, just like a field or method reference.

If you compile a class that uses Bread.rye, and then later add some other bread flavors at the beginning of the enumeration (pumpernickel and oatmeal), Bread.rye will now have a different numeric value. But you do not need to recompile any classes that use the enumeration. Even better, if you remove an enum constant that is actually being used by some other class (and you forget to recompile the class where it's used), the run-time library will issue an informative error message as soon as you use the now-removed name. This is a significant boost to making Java software more reliable.

Statements Updated for Enumerations

Two statements have been changed to make them work better with enumerations: the switch statement, and the for statement.

Using a for statement with enums

Language designers want to make it easy to express the concept "I*terate through all the values in this enumeration type*". There was much discussion about how to modify the "for" loop statement. It was done in a clever way—by adding support for iterating through all the values in any array.

Here is the new "get all elements of an array" syntax added to the "for" statement.

```
// for each Dog object in the dogsArray[] print out the dog's name
for (Dog dog : dogsArray ){
    dog.printName();
}
```

This style of "for" statement has become known as the "foreach" statement (which is really goofy, because there is no each or foreach keyword). The colon is read as "in", so the whole thing is read as "for each dog in dogsArray". The loop variable `dog` iterates through all the elements in the array starting at zero.

Zermelo-Fränkel set theory and you

Keywords don't have to be reserved words. So keywords don't have to be prohibited for use as identifiers. A compiler can look at the tokens around a keyword and decide whether it is being used as a keyword or an identifier. This is called *context-sensitive parsing*.

I would have preferred an actual keyword "in" rather than punctuation, but the colon has been used this way in Zermelo-Fränkel set theory by generations of mathematicans. It's just syntactic sugar. And if we've always done something this way, hey, don't not go for it.

Using the new foreach feature, we can now write a very brief program that echoes the arguments from the command line. The command line arguments (starting after the main class name) are passed to `main()` in the String array parameter. We print it out like this:

```
public class echo {
    public static void main( String[] args) {
        for (String i : args )
            System. out. println("arg = " + i);
    }
}
```

Compiling and running the program with a few arguments gives this:


```
javac -source 1.5 -target 1.5 echo. java

java echo test data here
arg = test
arg = data
arg = here
```

Note: The final decision on whether to have a "-source 1.5" compiler option and what it should look like had not been made when this book went to press. See the website `www.afu.com/jj6` for the latest information. You can set it up in a batch file so you don't have to keep typing it.

The same for loop syntax can be used for enums if you can somehow get the enum constants into an array. And that's exactly how it was done. Every enum type that you define automatically brings into existence a class method called `values()`.

The values() method for enums

The `values()` method is created on your behalf in every enum you define. It has a return value of an array, with successive elements containing the enum values. That can be used in for loops as the example shows:

```
for (Bread b : Bread.values() ) {

    // b iterates through all the Bread enum constants
    System.out.println("bread type = " + b);

}
```

In addition to the enum constants that you write, the enum type can contain other fields. The `values()` method is one of them. `Bread.values()` is a call to a class method belonging to the Bread enum. The method is invoked on the enum class type Bread, not on an instance, so we can conclude it must be a class method.

You can even iterate through a subrange of an enum type. The class java.util. EnumSet has a method called `range()` that takes various arguments, and returns a value suitable for use in the expanded for statement.

You would use it like this:

```
public class Example {
   enum Month {Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec }
   public static void main(String[] args) {

      for (Month m : java.util.EnumSet.range(Month.Jun, Month.Aug) )
                System.out.println("summer includes: " + m );
   }
}
```

When compiled and run, that prints:

```
summer includes: Jun
summer includes: Jul
summer includes: Aug
```

The method `range()` belonging to class EnumSet returns, not an array (as you might guess), but an object of type EnumSet. The "for" statement has been modified to accept an array *or* an EnumSet in this kind of loop (it can accept a Collection class too).

Modifying the "for" loopto allow a special type here was done as a concession to performance improvement. An EnumSet is a collection class that can hold multiple enum constants from any one enum type. They don't have to be a consecutive range of enum constants. If there are 64 or fewer enum constants in the set, it will store the set as a long word of bits, and not as an array of references. This makes it very fast to tell if an enum constant is in or out of the set, and very fast to iterate through a range of enum constants.

Using a switch statement with enums

You can now use an enum type in a "switch" statement (Java's name for the case statement used in other languages). Formerly, the switch expression had to be an int type. Here's an example that uses an enum in a switch statement.

```
Bread myLoaf = Bread.rye;
int price = 0;

switch (myLoaf) {
    case wholewheat: price = 190;
                     break;

    case     french: price = 200;
                     break;

     case ninegrain: price = 250;
                     break;

           case rye: price = 275;
                     break;
}

System.out.println( myLoaf + " costs " + price);
```

The "break" statement causes the flow of control to transfer to the end of the switch. Without that, program execution drops through into the next case clause. Running the program gives the output:

```
rye costs 275
```

The enum value in the case switch label must be an unqualified name. That is, you must write "rye" and not "Bread. rye". That (unfortunately) contrasts with an assignment to an enum variable. In an assignment:

```
myLoaf = Bread.rye;
```

you must use the qualified name Bread.rye, unless you import the type name.

Dropping the need to qualify names—import

We mentioned above that you must use the qualified name, such as Bread.rye, *unless you import the type name*. This section explains how to do that, using a wordy example from the digital clock in Chapter 2. That program had a statement saying:

```
this.setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
```

The statement is setting how the window should behave when the user clicks on the close icon on the title bar. There's a choice of four alternatives, ranging from "ignore it" to "terminate the program". The alternatives are represented by integer constants in class WindowConstants, which is part of package javax.swing. Here is the source code from the Java run-time library, defining those constants:

```
public static final int DO_NOTHING_ON_CLOSE = 0;
public static final int HIDE_ON_CLOSE = 1;
public static final int DISPOSE_ON_CLOSE = 2;
public static final int EXIT_ON_CLOSE = 3;
```

Those could be better expressed as an enum type, but this code was written long before Java supported enum types. It would be nice if the programmer using those constants didn't have to mention the lengthy names of the package and subpackage they come from.

That's exactly what the "import" statement gives you. The import statement has been part of Java from the beginning. It lets a programmer write this:

```
import java.swingx.*;   // import all classes in package java.swingx
```

or

```
import java.swingx.WindowConstants;  // import this class
```

The "import" statements (if any) appear at the top of the source file, right after the optional line that says what package this file belongs to. The imports apply to all the rest of the file. The import statement brings into your namespace one or more classes from a package. The import with the asterisk is meant to be like a wildcard in a file name. It means "import all classes from the package". So "import javax.swing.*;" means you can drop the "javax.swing" qualifier from class names from the javax.swing package.

The second version, importing a single class, means that you do not have to qualify that one class with its package name at each point you mention it.

Import is purely a compile time convenience feature. It does not make a class visible that was not visible before. It does not make any change to the amount of code that is linked into your executable. It just lets you refer to a class without qualifying it with the full name of the package it is in.

Either of these imports would allow us to cut down our "close window" statement to:

```
this.setDefaultCloseOperation( WindowConstants.EXIT_ON_CLOSE );
```

Using "import static"

New!

We are still required to mention the class name, and this is where the new feature of *static import* comes in. JDK 1.5 adds the ability to import the static members from a class, or the enum values from an enum type. It is analogous to the ordinary import statement, and it looks like this:

```
import static qualified_class_name.*;
```

or

```
import static qualified_enum_name.*;
```

So you can now write

```
import static javax.swing.WindowConstants.*;
   /* more code */
  this.setDefaultCloseOperation(EXIT_ON_CLOSE);
```

The *this* can be implicit, so what started out as

```
this.setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
```

can now be written as

```
import static javax.swing.WindowConstants.*;
  //  lots more code here
setDefaultCloseOperation(EXIT_ON_CLOSE);
```

Similarly, using enums and static import we can write:

```
import static mypackage.Bread.* ;
      // more code here

       Bread myLoaf = rye;
```

You can import just one enum value or static member, by appending its name instead of using the "*" to get all of them. Don't forget that to compile anything with the new JDK 1.5 features, you need to use these options on the command line:

```
javac -source 1.5 -target 1.5 sourcefiles.java
```

You can use import static on the System class to shorten the name of the utility that does console I/O. Before import static, I used to create an alias like this:

```
java.io.PrintStream o = System.out;
o.println("hello Bob");
```

Now you can "import static" instead.

```
import static java.lang.System.out; /* more code omitted */
out.println("hello Bob");
```

The compiler is quite capable of figuring out whether you are importing a static member or a classname within a package, so the "import static" keywords could have been left as just the old "import" keyword. It was considered better to remind the programmer that only static members can be imported, not instance members.

What if you import static WarmColors and Fruits? You can do this (they will need to be in a named package, not the default anonymous package used in the examples). But any place where you use one of the named constants whose name is duplicated, you will have to provide the full qualified name:

```
Fruit f = Fruit.peach;
Fruit f2 =      apple;  // unique name, so no qualifier needed.
```

At this point we've covered everything relating to enums that you'll see 99% of the time. There is some more material, but you can safely skip the rest of the chapter, and return when you have some specific enum construct to decode.

More Complicated Enumerated Types

Since enums are effectively classes, you can do pretty much everything with them that you can do with a class. In particular, you can provide one or more constructors for an enum type! That may seem a little weird to you (it does to me), because you never call an enum constructor anywhere in your code, or use the "new" operator to instantiate an enum. The definition of an enum brings the class enum constants into existence. You always work with static fields of an enum, not instances.

You might want to write a constructor when you have enumerations with a close relationship to numeric values. An example is an enumeration of hens' egg sizes,

shown in Table 6–1. In the U.S., these are the names and weights associated with eggs:

Table 6–1 U.S. names and weights associated with eggs

Name	weight per dozen
Jumbo	30 oz
Extra large	27 oz
Large	24 oz

Adding constructors to your enums

We're going to create an enum class called egg, with enum constants of jumbo, etc. You can tie arbitrary data to each enum constant by writing a constructor for the enum class. In other circumstances, a constructor has the same name as the class, so you'd expect it to be called egg here. And indeed it is, but that's not the full story. You declare an enum constant (jumbo, etc) and that *name declaration* is regarded as a *call* to your constructor. You may pass data values as arguments to the call. Arguments are passed in the usual way, by enclosing the comma-separated list in parentheses.

Putting it together, the enum now looks like this:

```
enum egg {
            // the enum constants, which "call" the constructor
            jumbo(30.0),
            extraLarge(27.0),
            large(24.0);

            egg(double w) {weight=w;} // constructor

            private double weight;

}
```

The beta JDK 1.5 compiler requires the enum constants to come before the constructor. There's no good reason for that and I filed it as a bug, but no word yet on whether Sun sees it the same way. As well as constructors, you can add practically any methods or data fields in an enum class. The "private" keyword makes a member inaccessible from outside the class. So you probably want to add this method to the enum to be able to retrieve the weight of an enum variable:

```
        double getWeight() { return this.weight; }
```

Here's a small main program that uses the enum, and prints out the weight for jumbo eggs.

```
public class bigegg {
    public static void main( String[] args) {
        egg e = egg.jumbo;
        double wt = e. getWeight();
          System.out.println( e + " eggs weigh "+ wt +" oz. per doz.");
    }
}
```

Running the code gives this output:

```
jumbo eggs weigh 30.0oz. per doz.
```

The language specification guarantees that all enum constants are unique. There may be two different reference variables pointing to Bread.rye, but there are never two different Bread.rye enum constants. That ensures programmers can compare enums using `e == Bread.rye` as well as `e.equals( Bread. rye)`.

You need to avoid declaring any members in your enums with names that duplicate those in java.lang.Enum. The box below has a list of most of the methods. The compiler will tell you if you make this mistake.

Predefined members that belong to every enum

The compiler automatically creates several other members in addition to `values()` in every enum class. These additional members help the enum do its work. The compiler will warn you if you re-use one of these names. Furthermore, each enum type is regarded as a child of class java.lang.Enum, and therefore inherits those methods from there too. Some of the methods in java.lang.Enum are:

```
public int compareTo( Enum e );
// returns negative, 0, positive for this earlier, equal, or later
// in declaration order than e

public int ordinal();
// returns the position of the enum constant in the
// declaration, first enum constant is at position 0

public static EnumType valueOf( java.lang.Class ec, String s );
// turns a String into its corresponding enum constant
// the ec argument is the class object of the enum type we are using

public boolean equals( Object o );
// returns true when this equals the enum specified by Object o.
```

Instance methods and enum variables

We mentioned back at the start of the chapter that enum types are implemented as classes, so you can add your own methods to them. Here's an example showing how you can do that:


```
enum Month {
     Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec;

     int days() {
        if ((this==Sep)|(this==Apr)|(this==Jun)|(this==Nov)) return 30;
        if (this==Feb)
          throw new UnsupportedOperationException("need leap year status");
        return 31;
     }
}
```

We added the method `days()` to enum Month. It returns a count of the number of days in the month, except for Feb when it will throw an exception. That's a lame response, but better than allowing the day count to be wrong. What this says is that the `days()` method should not be part of the enum Month (in real world code), because it does not have enough information to do its job.

You can declare an enum variable like this:

```
Month easterMonth;
```

You can use it like this:

```
// Easter is always between March 22 and April 25
easterMonth = calcMonthOfEaster(); // this method not shown!
if (easterMonth == Month.Mar )   /* do something*/ ;

if (easterMonth.days() != 31)    /* do something else */ ;
```

Constant-specific class bodies

There's only one more aspect of enums to cover, and it's something that you will encounter rarely in practice: a feature known as constant-specific class bodies. We've seen previously how enums are implemented in terms of classes, and how they can have their own constructors, methods and data, as well as the enum constants.

Going a step beyond this, each enum constant in an enum type can be given its own class body. You write it as the enum constant name followed by a block like this

```
{ body_of_a_class }
```

and anything that can appear in a class can go in that body.

This kind of class body attached to an enum constant is called a constant-specific class body, and it is regarded as an anonymous (cannot be referred to by a name) class that extends the enum type. When a child class extends a parent, it may provide its own version of a method with the same signature in the parent. This process of replacing a method is known as *overriding*. In particular, an instance method in a constant-specific class body will override an instance method of the same signature that belongs to the enum type. Phew!

Here's an example, building on the basic egg enum:

```
enum egg {

    large { /* this is a constant-specific class body*/ },
    extraLarge { /* so is this, and they are empty*/ },
    jumbo { /* yep, same here*/ };

    public double recipeEquivalent() { return 1.0; }
}
```

The example shows three enum constants, each with an empty constant-specific class body. The enum type also has an instance method called `recipeEquivalent()`.

As you may know, when a recipe calls for eggs, it always means large eggs[1]. We are going to override the enum type method `recipeEquivalent()` and provide a constant specific method that gives the "scaling factor" for jumbo and extra large eggs. To make your recipes come out right, you need to use fractions of eggs when using the bigger eggs. The method `recipeEquivalent()` provides the scaling factor for each egg size.

Referring back to table 6-1, by weight,

1 large egg == (1extra-large egg * 24/ 27) == (1 jumbo egg * 24/ 30).

That can be expressed in the code like this:

```
enum egg {
    large,
    extraLarge { public double recipeEquivalent(){ return 24.0/ 27.0; }},
    jumbo { public double recipeEquivalent(){ return 24.0/ 30.0; }};

    public double recipeEquivalent() { return 1.0; }
}
```

There are other ways to get the same effect of scaling different egg sizes. This example is meant to show how you might use a constant-specific class body. Here's a main program to test the feature:

```
public class eggtest {
    public static void main( String[] args) {
       System.out.println(" when the recipe calls for 1 egg, use:");
       for (egg e : egg.values() ) {
         System.out.printf("%f %s eggs %n", e.recipeEquivalent(), e );
       }
    }
}
```

1. If you didn't know this, I don't want to eat sponge cake at your house.

The result of compiling and running that code is:

```
when the recipe calls for 1 egg, use:
1.00 large eggs
0.88 extraLarge eggs
0.80 jumbo eggs
```

Note the use of the new `printf()` method. Chapter 17 has the full details. Remember to add an option on the command line to compile these new features:

```
javac -source 1.5 egg.java eggtest.java
```

Finally, if your enums start to sprout constructors, methods, constant specific bodies and instance data, maybe your enum type should just be a class in the first place.

Some Light Relief—The Haunted Zen Garden of Apple

I've always made a point of exploring new and unfamiliar places, particularly when I'm trying a new shortcut and forgot to bring the map with me. My colleague Lefty is just the opposite. Lefty (who I taught to count on his fingers in binary—you can count up to 1023 that way) likes to stick to paths so well beaten they are practically pulverized.

Lefty and I were on the Apple Computer campus in Cupertino, California. If you know the area, we were in the Bandley 3 building, and late for a meeting in nearby De Anza 7. You can almost see these two buildings from each other, so I was practically sure that if we jogged through the parking lots as the crow flies, we'd get to De Anza 7 in the shortest amount of time.

“I hope you're not trying one of your shortcuts,” accused Lefty as he sprinted behind me, vaulting over fences and bushes. “Don’t be ridiculous!” I retorted. We would have arrived almost on time too, if my shortcut hadn't taken us through a small and mysterious grove of trees at the side of the De Anza 7 parking lot. Lefty ran into the grove, then straight into a large waist-high stone.

Figure 6–2 The Haunted Zen Garden of Apple (De Anza 7 in background)

The big southpaw went down with a curse that would curl the hair of a software architect. While he threshed on the ground, mouthing incoherent criticism of me and shortcuts, I examined the stone with interest. It appeared to be a tombstone covered in Japanese writing.

I tried to redirect his attention to the mysterious stone "Check this out Lefty," I noted, "It's just like that slab on Jupiter in 2001!". His reply cannot be printed in a text that family members may read. Looking around more widely, I was very surprised to see that the grove was an abandoned Zen garden. My first thought was to wonder if the tombstone meant the Zen garden was haunted, cursed, or at least full of bugs.

A mystery like this needs a solution. Even if it doesn't, it's going to get one. The first theory was that the haunted Zen garden had been created by an Apple executive who'd fallen prey to the "Japan does everything better" management fad of the month. But nobody at Apple wanted to confess to knowing anything about *that*.

To cut a long story short, the true story became clear when I got the stone inscription translated. It reads:

"This place has been cultivated by Ernest and Lillian Wanaka as part of the Cupertino Nursery since 1947. The Wanakas have dreamed of creating an elegant Japanese Garden at this tranquil land that can be viewed from their residence. At last they decided to start working on the dream. The completed garden was opened in 1954 as a place of contemplation for Cupertino citizens."

The rest of the story soon emerged. The property all around was formerly owned by the Japanese Wanaka family of Cupertino, who lived on it and tended it as a nursery. When Apple bought the land from them, a clause in the contract stipulated that the Zen garden had to be left in place in perpetuity. (Lefty gasped at this bit, and pointed out that Apple could park another 15 cars in the space the garden occupied. Lefty is a very practical person.) The abandoned Zen garden is more than 50 years old, and predates the Macintosh, Apple, and Silicon Valley itself.

The abandoned Zen garden remains there today, at the corner of De Anza 7. That building was at one time the hub of executive power at Apple. As some used to joke, De Anza 7 was "where the rubber meets the air". The stone that tripped Lefty wasn't a tombstone on unhallowed ground; it was a dedication in a garden. If that land is haunted, it is only by the spirits of past Apple executives like Sculley, Spindler, Yocom, Graziano, Sullivan, Stead, and Eisenstat.

"Why doesn't Apple pay a gardening service to keep up the garden?" Lefty asked.

I thought about how you can't pay someone to maintain a Zen garden for you. That would be karma-lama-ding-dong. The point of a Zen garden is that you maintain it yourself and rake the sand around the rocks according to your meditations of the day. Today all the Zen is in the Apple products, not in the parking lots.

"There are no shortcuts to Zen, Lefty" I loftily replied.