

---

## Enums and Annotations

**I**N release 1.5, two families of reference types were added to the language: a new kind of class called an *enum type*, and a new kind of interface called an *annotation type*. This chapter discusses best practices for using these new type families.

### Item 30: Use enums instead of int constants

An *enumerated type* is a type whose legal values consist of a fixed set of constants, such as the seasons of the year, the planets in the solar system, or the suits in a deck of playing cards. Before enum types were added to the language, a common pattern for representing enumerated types was to declare a group of named int constants, one for each member of the type:

```
// The int enum pattern - severely deficient!
public static final int APPLE_FUJI      = 0;
public static final int APPLE_PIPPIN    = 1;
public static final int APPLE_GRANNY_SMITH = 2;

public static final int ORANGE_NAVEL    = 0;
public static final int ORANGE_TEMPLE    = 1;
public static final int ORANGE_BLOOD    = 2;
```

This technique, known as the *int enum pattern*, has many shortcomings. It provides nothing in the way of type safety and little in the way of convenience.



The compiler won't complain if you pass an apple to a method that expects an orange, compare apples to oranges with the == operator, or worse:

```
// Tasty citrus flavored applesauce!
int i = (APPLE_FUJI - ORANGE_TEMPLE) / APPLE_PIPPIN;
```

Note that the name of each apple constant is prefixed with `APPLE_` and the name of each orange constant is prefixed with `ORANGE_`. This is because Java doesn't provide namespaces for `int` enum groups. Prefixes prevent name clashes when two `int` enum groups have identically named constants.

Programs that use the `int` enum pattern are brittle. Because `int` enums are compile-time constants, they are compiled into the clients that use them. If the `int` associated with an enum constant is changed, its clients must be recompiled. If they aren't, they will still run, but their behavior will be undefined.

There is no easy way to translate `int` enum constants into printable strings. If you print such a constant or display it from a debugger, all you see is a number, which isn't very helpful. There is no reliable way to iterate over all the `int` enum constants in a group, or even to obtain the size of an `int` enum group.

You may encounter a variant of this pattern in which `String` constants are used in place of `int` constants. This variant, known as the *String enum pattern*, is even less desirable. While it does provide printable strings for its constants, it can lead to performance problems because it relies on string comparisons. Worse, it can lead naive users to hard-code string constants into client code instead of using field names. If such a hard-coded string constant contains a typographical error, it will escape detection at compile time and result in bugs at runtime.

Luckily, as of release 1.5, the language provides an alternative that avoids the shortcomings of the `int` and `string` enum patterns and provides many added benefits. It is the (JLS, 8.9). Here's how it looks in its simplest form:

```
public enum Apple { FUJI, PIPPIN, GRANNY_SMITH }
public enum Orange { NAVEL, TEMPLE, BLOOD }
```

On the surface, these enum types may appear similar to those of other languages, such as C, C++, and C#, but appearances are deceiving. Java's enum types are full-fledged classes, far more powerful than their counterparts in these other languages, where enums are essentially `int` values.



The basic idea behind Java's enum types is simple: they are classes that export one instance for each enumeration constant via a public static final field. Enum types are effectively final, by virtue of having no accessible constructors. Because clients can neither create instances of an enum type nor extend it, there can be no instances but the declared enum constants. In other words, enum types are instance-controlled (page 6). They are a generalization of singletons (Item 3), which are essentially single-element enums. For readers familiar with the first edition of this book, enum types provide linguistic support for the *typesafe enum* pattern [Bloch01, Item 21].

Enums provide compile-time type safety. If you declare a parameter to be of type `Apple`, you are guaranteed that any non-null object reference passed to the parameter is one of the three valid `Apple` values. Attempts to pass values of the wrong type will result in compile-time errors, as will attempts to assign an expression of one enum type to a variable of another, or to use the `==` operator to compare values of different enum types.

Enum types with identically named constants coexist peacefully because each type has its own namespace. You can add or reorder constants in an enum type without recompiling its clients because the fields that export the constants provide a layer of insulation between an enum type and its clients: the constant values are not compiled into the clients as they are in the `int` enum pattern. Finally, you can translate enums into printable strings by calling their `toString` method.

In addition to rectifying the deficiencies of `int` enums, enum types let you add arbitrary methods and fields and implement arbitrary interfaces. They provide high-quality implementations of all the `Object` methods (Chapter 3), they implement `Comparable` (Item 12) and `Serializable` (Chapter 11), and their serialized form is designed to withstand most changes to the enum type.

So why would you want to add methods or fields to an enum type? For starters, you might want to associate data with its constants. Our `Apple` and `Orange` types, for example, might benefit from a method that returns the color of the fruit, or one that returns an image of it. You can augment an enum type with any method that seems appropriate. An enum type can start life as a simple collection of enum constants and evolve over time into a full-featured abstraction.

For a nice example of a rich enum type, consider the eight planets of our solar system. Each planet has a mass and a radius, and from these two attributes you can compute its surface gravity. This in turn lets you compute the weight of an object on the planet's surface, given the mass of the object. Here's how this enum looks. The numbers in parentheses after each enum constant are parameters that are passed to its constructor. In this case, they are the planet's mass and radius:

```
// Enum type with data and behavior
public enum Planet {
    MERCURY(3.302e+23, 2.439e6),
    VENUS (4.869e+24, 6.052e6),
    EARTH (5.975e+24, 6.378e6),
    MARS (6.419e+23, 3.393e6),
    JUPITER(1.899e+27, 7.149e7),
    SATURN (5.685e+26, 6.027e7),
    URANUS (8.683e+25, 2.556e7),
    NEPTUNE(1.024e+26, 2.477e7);
```



```

private final double mass;           // In kilograms
private final double radius;         // In meters
private final double surfaceGravity; // In m / s^2

// Universal gravitational constant in m^3 / kg s^2
private static final double G = 6.67300E-11;

// Constructor
Planet(double mass, double radius) {
    this.mass = mass;
    this.radius = radius;
    surfaceGravity = G * mass / (radius * radius);
}

public double mass()           { return mass; }
public double radius()         { return radius; }
public double surfaceGravity() { return surfaceGravity; }

public double surfaceWeight(double mass) {
    return mass * surfaceGravity; // F = ma
}
}

```


It is easy to write a rich enum type such as Planet. **To associate data with enum constants, declare instance fields and write a constructor that takes the data and stores it in the fields.** Enums are by their nature immutable, so **all fields should be final** (Item 15). They can be public, but it is better to make them private and provide public accessors (Item 14). In the case of Planet, the constructor also computes and stores the surface gravity, but this is just an optimization. The gravity could be recomputed from the mass and radius each time it was used by the surfaceWeight method, which takes an object's mass and returns its weight on the planet represented by the constant.

While the Planet enum is simple, it is surprisingly powerful. Here is a short program that takes the earth-weight of an object (in any unit) and prints a nice table of the object's weight on all eight planets (in the same unit):

```

public class WeightTable {
    public static void main(String[] args) {
        double earthWeight = Double.parseDouble(args[0]);
        double mass = earthWeight / Planet.EARTH.surfaceGravity();
        for (Planet p : Planet.values())
            System.out.printf("Weight on %s is %f%n",
                               p, p.surfaceWeight(mass));
    }
}

```



Note that `Planet`, like all enums, has a static `values` method that returns an array of its values in the order they were declared. Note also that the `toString` method returns the declared name of each enum value, enabling easy printing by `println` and `printf`. If you're dissatisfied with this string representation, you can change it by overriding the `toString` method. Here is the result of running our little `WeightTable` program with the command line argument `175`:

```
Weight on MERCURY is 66.133672
Weight on VENUS is 158.383926
Weight on EARTH is 175.000000
Weight on MARS is 66.430699
Weight on JUPITER is 442.693902
Weight on SATURN is 186.464970
Weight on URANUS is 158.349709
Weight on NEPTUNE is 198.846116
```

If this is the first time you've seen Java's `printf` method in action, note that it differs from C's in that you use `%n` where you'd use `\n` in C.

Some behaviors associated with enum constants may need to be used only from within the class or package in which the enum is defined. Such behaviors are best implemented as private or package-private methods. Each constant then carries with it a hidden collection of behaviors that allows the class or package containing the enum to react appropriately when presented with the constant. Just as with other classes, unless you have a compelling reason to expose an enum method to its clients, declare it private or, if need be, package-private (Item 13).

If an enum is generally useful, it should be a top-level class; if its use is tied to a specific top-level class, it should be a member class of that top-level class (Item 22). For example, the `java.math.RoundingMode` enum represents a rounding mode for decimal fractions. These rounding modes are used by the `BigDecimal` class, but they provide a useful abstraction that is not fundamentally tied to `BigDecimal`. By making `RoundingMode` a top-level enum, the library designers encourage any programmer who needs rounding modes to reuse this enum, leading to increased consistency across APIs.

The techniques demonstrated in the `Planet` example are sufficient for most enum types, but sometimes you need more. There is different data associated with each `Planet` constant, but sometimes you need to associate fundamentally different *behavior* with each constant. For example, suppose you are writing an enum type to represent the operations on a basic four-function calculator, and you want

to provide a method to perform the arithmetic operation represented by each constant. One way to achieve this is to switch on the value of the enum:

```
// Enum type that switches on its own value - questionable
public enum Operation {
    PLUS, MINUS, TIMES, DIVIDE;

    // Do the arithmetic op represented by this constant
    double apply(double x, double y) {
        switch(this) {
            case PLUS:    return x + y;
            case MINUS:   return x - y;
            case TIMES:   return x * y;
            case DIVIDE:  return x / y;
        }
        throw new AssertionError("Unknown op: " + this);
    }
}
```

This code works, but is isn't very pretty. It won't compile without the `throw` statement because the end of the method is technically reachable, even though it will never be reached [JLS, 14.2.1]. Worse, the code is fragile. If you add a new enum constant but forget to add a corresponding case to the `switch`, the enum will still compile, but it will fail at runtime when you try to apply the new operation.

Luckily, there is a better way to associate a different behavior with each enum constant: declare an abstract `apply` method in the enum type, and override it with a concrete method for each constant in a *constant-specific class body*. Such methods are known as *constant-specific method implementations*:

```
// Enum type with constant-specific method implementations
public enum Operation {
    PLUS { double apply(double x, double y){return x + y;} },
    MINUS { double apply(double x, double y){return x - y;} },
    TIMES { double apply(double x, double y){return x * y;} },
    DIVIDE { double apply(double x, double y){return x / y;} };

    abstract double apply(double x, double y);
}
```

If you add a new constant to the second version of `Operation`, it is unlikely that you'll forget to provide an `apply` method, as the method immediately follows each constant declaration. In the unlikely event that you do forget, the compiler will remind you, as abstract methods in an enum type must be overridden with concrete methods in all of its constants.

Constant-specific method implementations can be combined with constant-specific data. For example, here is a version of `Operation` that overrides the `toString` method to return the symbol commonly associated with the operation:

```
// Enum type with constant-specific class bodies and data
public enum Operation {
    PLUS("+") {
        double apply(double x, double y) { return x + y; }
    },
    MINUS("-") {
        double apply(double x, double y) { return x - y; }
    },
    TIMES("*") {
        double apply(double x, double y) { return x * y; }
    },
    DIVIDE("/") {
        double apply(double x, double y) { return x / y; }
    };
    private final String symbol;
    Operation(String symbol) { this.symbol = symbol; }
    @Override public String toString() { return symbol; }

    abstract double apply(double x, double y);
}
```

In some cases, overriding `toString` in an enum is very useful. For example, the `toString` implementation above makes it easy to print arithmetic expressions, as demonstrated by this little program:

```
public static void main(String[] args) {
    double x = Double.parseDouble(args[0]);
    double y = Double.parseDouble(args[1]);
    for (Operation op : Operation.values())
        System.out.printf("%f %s %f = %f\n",
                           x, op, y, op.apply(x, y));
}
```

Running this program with 2 and 4 as command line arguments produces the following output:

```
2.000000 + 4.000000 = 6.000000
2.000000 - 4.000000 = -2.000000
2.000000 * 4.000000 = 8.000000
2.000000 / 4.000000 = 0.500000
```

Enum types have an automatically generated `valueOf(String)` method that translates a constant's name into the constant itself. If you override the `toString` method in an enum type, consider writing a `fromString` method to translate the custom string representation back to the corresponding enum. The following code (with the type name changed appropriately) will do the trick for any enum, so long as each constant has a unique string representation:

```
// Implementing a fromString method on an enum type
private static final Map<String, Operation> stringToEnum
    = new HashMap<String, Operation>();
static { // Initialize map from constant name to enum constant
    for (Operation op : values())
        stringToEnum.put(op.toString(), op);
}
// Returns Operation for string, or null if string is invalid
public static Operation fromString(String symbol) {
    return stringToEnum.get(symbol);
}
```

Note that the `Operation` constants are put into the `stringToEnum` map from a static block that runs after the constants have been created. Trying to make each constant put itself into the map from its own constructor would cause a compilation error. This is a good thing, because it would cause a `NullPointerException` if it were legal. Enum constructors aren't permitted to access the enum's static fields, except for compile-time constant fields. This restriction is necessary because these static fields have not yet been initialized when the constructors run.

A disadvantage of constant-specific method implementations is that they make it harder to share code among enum constants. For example, consider an enum representing the days of the week in a payroll package. This enum has a method that calculates a worker's pay for that day given the worker's base salary (per hour) and the number of hours worked on that day. On the five weekdays, any time worked in excess of a normal shift generates overtime pay; on the two weekend days, all work generates overtime pay. With a `switch` statement, it's easy to do this calculation by applying multiple case labels to each of two code fragments. For brevity's sake, the code in this example uses `double`, but note that `double` is *not* an appropriate data type for a payroll application (Item 48):

```
// Enum that switches on its value to share code - questionable
enum PayrollDay {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
    SATURDAY, SUNDAY;
    private static final int HOURS_PER_SHIFT = 8;
```



```

double pay(double hoursWorked, double payRate) {
    double basePay = hoursWorked * payRate;

    double overtimePay;    // Calculate overtime pay
    switch(this) {
        case SATURDAY: case SUNDAY:
            overtimePay = hoursWorked * payRate / 2;
        default: // Weekdays
            overtimePay = hoursWorked <= HOURS_PER_SHIFT ?
                0 : (hoursWorked - HOURS_PER_SHIFT) * payRate / 2;
            break;
    }

    return basePay + overtimePay;
}

```

This code is undeniably concise, but it is dangerous from a maintenance perspective. Suppose you add an element to the enum, perhaps a special value to represent a vacation day, but forget to add a corresponding case to the switch statement. The program will still compile, but the pay method will silently pay the worker the same amount for a vacation day as for an ordinary weekday.

To perform the pay calculation safely with constant-specific method implementations, you would have to duplicate the overtime pay computation for each constant, or move the computation into two helper methods (one for weekdays and one for weekend days) and invoke the appropriate helper method from each constant. Either approach would result in a fair amount of boilerplate code, substantially reducing readability and increasing the opportunity for error.

The boilerplate could be reduced by replacing the abstract `overtimePay` method on `PayrollDay` with a concrete method that performs the overtime calculation for weekdays. Then only the weekend days would have to override the method. But this would have the same disadvantage as the switch statement: if you added another day without overriding the `overtimePay` method, you would silently inherit the weekday calculation.

What you really want is to be *forced* to choose an overtime pay strategy each time you add an enum constant. Luckily, there is a nice way to achieve this. The idea is to move the overtime pay computation into a private nested enum, and to pass an instance of this *strategy enum* to the constructor for the `PayrollDay` enum. The `PayrollDay` enum then delegates the overtime pay calculation to the strategy enum, eliminating the need for a switch statement or constant-specific method

implementation in `PayrollDay`. While this pattern is less concise than the switch statement, it is safer and more flexible:

```
// The strategy enum pattern
enum PayrollDay {
    MONDAY(PayType.WEEKDAY), TUESDAY(PayType.WEEKDAY),
    WEDNESDAY(PayType.WEEKDAY), THURSDAY(PayType.WEEKDAY),
    FRIDAY(PayType.WEEKDAY),
    SATURDAY(PayType.WEEKEND), SUNDAY(PayType.WEEKEND);

    private final PayType payType;
    PayrollDay(PayType payType) { this.payType = payType; }

    double pay(double hoursWorked, double payRate) {
        return payType.pay(hoursWorked, payRate);
    }
}

// The strategy enum type
private enum PayType {
    WEEKDAY {
        double overtimePay(double hours, double payRate) {
            return hours <= HOURS_PER_SHIFT ? 0 :
                (hours - HOURS_PER_SHIFT) * payRate / 2;
        }
    },
    WEEKEND {
        double overtimePay(double hours, double payRate) {
            return hours * payRate / 2;
        }
    };
    private static final int HOURS_PER_SHIFT = 8;

    abstract double overtimePay(double hrs, double payRate);

    double pay(double hoursWorked, double payRate) {
        double basePay = hoursWorked * payRate;
        return basePay + overtimePay(hoursWorked, payRate);
    }
}
}
```

If switch statements on enums are not a good choice for implementing constant-specific behavior on enums, what *are* they good for? **Switches on enums are good for augmenting external enum types with constant-specific behavior.** For example, suppose the `Operation` enum is not under your control, and you

wish it had an instance method to return the inverse of each operation. You can simulate the effect with the following static method:

```
// Switch on an enum to simulate a missing method
public static Operation inverse(Operation op) {
    switch(op) {
        case PLUS:    return Operation.MINUS;
        case MINUS:   return Operation.PLUS;
        case TIMES:   return Operation.DIVIDE;
        case DIVIDE:  return Operation.TIMES;
        default:      throw new AssertionError("Unknown op: " + op);
    }
}
```

Enums are, generally speaking, comparable in performance to `int` constants. A minor performance disadvantage of enums over `int` constants is that there is a space and time cost to load and initialize enum types. Except on resource-constrained devices, such as cell phones and toasters, this is unlikely to be noticeable in practice.

So when should you use enums? Anytime you need a fixed set of constants. Of course, this includes “natural enumerated types,” such as the planets, the days of the week, and the chess pieces. But it also includes other sets for which you know all the possible values at compile time, such as choices on a menu, operation codes, and command line flags. It is not necessary that the set of constants in an enum type stay fixed for all time. The enum feature was specifically designed to allow for binary compatible evolution of enum types.

In summary, the advantages of enum types over `int` constants are compelling. Enums are far more readable, safer, and more powerful. Many enums require no explicit constructors or members, but many others benefit from associating data with each constant and providing methods whose behavior is affected by this data. Far fewer enums benefit from associating multiple behaviors with a single method. In this relatively rare case, prefer constant-specific methods to enums that switch on their own values. Consider the strategy enum pattern if multiple enum constants share common behaviors.