CHAPTER 6

Enums and Annotations

JAVA supports two special-purpose families of reference types: a kind of class called an *enum type,* and a kind of interface called an *annotation type*. This chapter discusses best practices for using these type families.

Item 34: 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 expressive power. 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, for example between `ELEMENT_MERCURY` and `PLANET_MERCURY`.

Programs that use `int` enums are brittle. Because `int` enums are *constant variables* [JLS, 4.12.4], their `int` values are compiled into the clients that use them [JLS, 13.1]. If the value associated with an `int` enum is changed, its clients must be recompiled. If not, the clients will still run, but their behavior will be incorrect.

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 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. Also, it might lead to performance problems, because it relies on string comparisons.

Luckily, Java provides an alternative that avoids all the shortcomings of the `int` and `string` enum patterns and provides many added benefits. It is the *enum type* [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.

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: constant values are not compiled into the clients as they are in the `int` enum patterns. 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 14) and `Serializable` (Chapter 12), 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 17). Fields can be public, but it is better to make them private and provide public accessors (Item 16). 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 `WeightTable` program (which doesn't override `toString`) with the command line argument 185:

```
Weight on MERCURY is 69.912739
Weight on VENUS is 167.434436
Weight on EARTH is 185.000000
Weight on MARS is 70.226739
Weight on JUPITER is 467.990696
Weight on SATURN is 197.120111
Weight on URANUS is 167.398264
Weight on NEPTUNE is 210.208751
```

Until 2006, two years after enums were added to Java, Pluto was a planet. This raises the question "what happens when you remove an element from an enum type?" The answer is that any client program that doesn't refer to the removed element will continue to work fine. So, for example, our `WeightTable` program would simply print a table with one fewer row. And what of a client program that refers to the removed element (in this case, `Planet.Pluto`)? If you recompile the client program, the compilation will fail with a helpful error message at the line that refers to the erstwhile planet; if you fail to recompile the client, it will throw a helpful exception from this line at runtime. This is the best behavior you could hope for, far better than what you'd get with the int enum pattern.

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 15).

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 24). 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 operation represented by this constant
    public 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 it 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, 14.21]. 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  {public double apply(double x, double y){return x + y;}},
  MINUS {public double apply(double x, double y){return x - y;}},
  TIMES {public double apply(double x, double y){return x * y;}},
  DIVIDE{public double apply(double x, double y){return x / y;}};

  public 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, because the method immediately follows each constant declaration. In the unlikely event that you do forget, the compiler will remind you because 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("+") {
        public double apply(double x, double y) { return x + y; }
    },
    MINUS("-") {
        public double apply(double x, double y) { return x - y; }
    },
    TIMES("*") {
        public double apply(double x, double y) { return x * y; }
    },
    DIVIDE("/") {
        public 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; }

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

The `toString` implementation shown 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 =
        Stream.of(values()).collect(
            toMap(Object::toString, e -> e));

// Returns Operation for string, if any
public static Optional<Operation> fromString(String symbol) {
    return Optional.ofNullable(stringToEnum.get(symbol));
}
```

Note that the `Operation` constants are put into the `stringToEnum` map from a static field initialization that runs after the enum constants have been created. The previous code uses a stream (Chapter 7) over the array returned by the `values()` method; prior to Java 8, we would have created an empty hash map and iterated over the values array inserting the string-to-enum mappings into the map, and you can still do it that way if you prefer. But note that attempting to have each constant put itself into a map from its own constructor does *not* work. It would cause a compilation error, which is good thing because if it were legal, it would cause a `NullPointerException` at runtime. Enum constructors aren't permitted to access the enum's static fields, with the exception of constant variables (Item 34). This restriction is necessary because static fields have not yet been initialized when enum constructors run. A special case of this restriction is that enum constants cannot access one another from their constructors.

Also note that the `fromString` method returns an `Optional<String>`. This allows the method to indicate that the string that was passed in does not represent a valid operation, and it forces the client to confront this possibility (Item 55).

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 minutes 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:

```
// Enum that switches on its value to share code - questionable
enum PayrollDay {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
    SATURDAY, SUNDAY;

    private static final int MINS_PER_SHIFT = 8 * 60;

    int pay(int minutesWorked, int payRate) {
        int basePay = minutesWorked * payRate;

        int overtimePay;
        switch(this) {
          case SATURDAY: case SUNDAY: // Weekend
            overtimePay = basePay / 2;
            break;
          default: // Weekday
            overtimePay = minutesWorked <= MINS_PER_SHIFT ?
              0 : (minutesWorked - MINS_PER_SHIFT) * payRate / 2;
        }

        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, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
    SATURDAY(PayType.WEEKEND), SUNDAY(PayType.WEEKEND);

    private final PayType payType;

    PayrollDay(PayType payType) { this.payType = payType; }
    PayrollDay() { this(PayType.WEEKDAY); }  // Default

    int pay(int minutesWorked, int payRate) {
        return payType.pay(minutesWorked, payRate);
    }

    // The strategy enum type
    private enum PayType {
        WEEKDAY {
            int overtimePay(int minsWorked, int payRate) {
                return minsWorked <= MINS_PER_SHIFT ? 0 :
                  (minsWorked - MINS_PER_SHIFT) * payRate / 2;
            }
        },
        WEEKEND {
            int overtimePay(int minsWorked, int payRate) {
                return minsWorked * payRate / 2;
            }
        };

        abstract int overtimePay(int mins, int payRate);
        private static final int MINS_PER_SHIFT = 8 * 60;

        int pay(int minsWorked, int payRate) {
            int basePay = minsWorked * payRate;
            return basePay + overtimePay(minsWorked, 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 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 could 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);
    }
}
```

You should also use this technique on enum types that *are* under your control if a method simply doesn't belong in the enum type. The method may be required for some use but is not generally useful enough to merit inclusion in the enum type.

Enums are, generally speaking, comparable in performance to `int` constants. A minor performance disadvantage of enums is that there is a space and time cost to load and initialize enum types, but it is unlikely to be noticeable in practice.

So when should you use enums? **Use enums any time you need a set of constants whose members are known at compile time.** 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 more readable, safer, and more powerful. Many enums require no explicit constructors or members, but others benefit from associating data with each constant and providing methods whose behavior is affected by this data. 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 some, but not all, enum constants share common behaviors.

Item 35: Use instance fields instead of ordinals

Many enums are naturally associated with a single `int` value. All enums have an `ordinal` method, which returns the numerical position of each enum constant in its type. You may be tempted to derive an associated `int` value from the ordinal:

```
// Abuse of ordinal to derive an associated value - DON'T DO THIS
public enum Ensemble {
    SOLO,   DUET,   TRIO, QUARTET, QUINTET,
    SEXTET, SEPTET, OCTET, NONET,  DECTET;

    public int numberOfMusicians() { return ordinal() + 1; }
}
```

While this enum works, it is a maintenance nightmare. If the constants are reordered, the `numberOfMusicians` method will break. If you want to add a second enum constant associated with an `int` value that you've already used, you're out of luck. For example, it might be nice to add a constant for *double quartet*, which, like an octet, consists of eight musicians, but there is no way to do it.

Also, you can't add a constant for an `int` value without adding constants for all intervening `int` values. For example, suppose you want to add a constant representing a *triple quartet*, which consists of twelve musicians. There is no standard term for an ensemble consisting of eleven musicians, so you are forced to add a dummy constant for the unused `int` value (`11`). At best, this is ugly. If many `int` values are unused, it's impractical.

Luckily, there is a simple solution to these problems. **Never derive a value associated with an enum from its ordinal; store it in an instance field instead:**

```
public enum Ensemble {
    SOLO(1), DUET(2), TRIO(3), QUARTET(4), QUINTET(5),
    SEXTET(6), SEPTET(7), OCTET(8), DOUBLE_QUARTET(8),
    NONET(9), DECTET(10), TRIPLE_QUARTET(12);

    private final int numberOfMusicians;
    Ensemble(int size) { this.numberOfMusicians = size; }
    public int numberOfMusicians() { return numberOfMusicians; }
}
```

The `Enum` specification has this to say about `ordinal`: "Most programmers will have no use for this method. It is designed for use by general-purpose enum-based data structures such as `EnumSet` and `EnumMap`." Unless you are writing code with this character, you are best off avoiding the `ordinal` method entirely.

Item 36: Use `EnumSet` instead of bit fields

If the elements of an enumerated type are used primarily in sets, it is traditional to use the `int` enum pattern (Item 34), assigning a different power of 2 to each constant:

```
// Bit field enumeration constants - OBSOLETE!
public class Text {
    public static final int STYLE_BOLD          = 1 << 0;  // 1
    public static final int STYLE_ITALIC        = 1 << 1;  // 2
    public static final int STYLE_UNDERLINE     = 1 << 2;  // 4
    public static final int STYLE_STRIKETHROUGH = 1 << 3;  // 8

    // Parameter is bitwise OR of zero or more STYLE_ constants
    public void applyStyles(int styles) { ... }
}
```

This representation lets you use the bitwise `OR` operation to combine several constants into a set, known as a *bit field*:

```
text.applyStyles(STYLE_BOLD | STYLE_ITALIC);
```

The bit field representation also lets you perform set operations such as union and intersection efficiently using bitwise arithmetic. But bit fields have all the disadvantages of `int` enum constants and more. It is even harder to interpret a bit field than a simple `int` enum constant when it is printed as a number. There is no easy way to iterate over all of the elements represented by a bit field. Finally, you have to predict the maximum number of bits you'll ever need at the time you're writing the API and choose a type for the bit field (typically `int` or `long`) accordingly. Once you've picked a type, you can't exceed its width (32 or 64 bits) without changing the API.

Some programmers who use enums in preference to `int` constants still cling to the use of bit fields when they need to pass around sets of constants. There is no reason to do this, because a better alternative exists. The `java.util` package provides the `EnumSet` class to efficiently represent sets of values drawn from a single enum type. This class implements the `Set` interface, providing all of the richness, type safety, and interoperability you get with any other `Set` implementation. But internally, each `EnumSet` is represented as a bit vector. If the underlying enum type has sixty-four or fewer elements—and most do—the entire `EnumSet` is represented with a single `long`, so its performance is comparable to that of a bit field. Bulk operations, such as `removeAll` and `retainAll`, are implemented using bit-

wise arithmetic, just as you'd do manually for bit fields. But you are insulated from the ugliness and error-proneness of manual bit twiddling: the `EnumSet` does the hard work for you.

Here is how the previous example looks when modified to use enums and enum sets instead of bit fields. It is shorter, clearer, and safer:

```
// EnumSet - a modern replacement for bit fields
public class Text {
    public enum Style { BOLD, ITALIC, UNDERLINE, STRIKETHROUGH }

    // Any Set could be passed in, but EnumSet is clearly best
    public void applyStyles(Set<Style> styles) { ... }
}
```

Here is client code that passes an `EnumSet` instance to the `applyStyles` method. The `EnumSet` class provides a rich set of static factories for easy set creation, one of which is illustrated in this code:

```
text.applyStyles(EnumSet.of(Style.BOLD, Style.ITALIC));
```

Note that the `applyStyles` method takes a `Set<Style>` rather than an `EnumSet<Style>`. While it seems likely that all clients would pass an `EnumSet` to the method, it is generally good practice to accept the interface type rather than the implementation type (Item 64). This allows for the possibility of an unusual client to pass in some other `Set` implementation.

In summary, **just because an enumerated type will be used in sets, there is no reason to represent it with bit fields.** The `EnumSet` class combines the conciseness and performance of bit fields with all the many advantages of enum types described in Item 34. The one real disadvantage of `EnumSet` is that it is not, as of Java 9, possible to create an immutable `EnumSet`, but this will likely be remedied in an upcoming release. In the meantime, you can wrap an `EnumSet` with `Collections.unmodifiableSet`, but conciseness and performance will suffer.

Item 37: Use `EnumMap` instead of ordinal indexing

Occasionally you may see code that uses the `ordinal` method (Item 35) to index into an array or list. For example, consider this simplistic class meant to represent a plant:

```
class Plant {
    enum LifeCycle { ANNUAL, PERENNIAL, BIENNIAL }

    final String name;
    final LifeCycle lifeCycle;

    Plant(String name, LifeCycle lifeCycle) {
        this.name = name;
        this.lifeCycle = lifeCycle;
    }

    @Override public String toString() {
        return name;
    }
}
```

Now suppose you have an array of plants representing a garden, and you want to list these plants organized by life cycle (annual, perennial, or biennial). To do this, you construct three sets, one for each life cycle, and iterate through the garden, placing each plant in the appropriate set. Some programmers would do this by putting the sets into an array indexed by the life cycle's ordinal:

```
// Using ordinal() to index into an array - DON'T DO THIS!
Set<Plant>[] plantsByLifeCycle =
    (Set<Plant>[]) new Set[Plant.LifeCycle.values().length];
for (int i = 0; i < plantsByLifeCycle.length; i++)
    plantsByLifeCycle[i] = new HashSet<>();

for (Plant p : garden)
    plantsByLifeCycle[p.lifeCycle.ordinal()].add(p);

// Print the results
for (int i = 0; i < plantsByLifeCycle.length; i++) {
    System.out.printf("%s: %s%n",
        Plant.LifeCycle.values()[i], plantsByLifeCycle[i]);
}
```

This technique works, but it is fraught with problems. Because arrays are not compatible with generics (Item 28), the program requires an unchecked cast and

will not compile cleanly. Because the array does not know what its index represents, you have to label the output manually. But the most serious problem with this technique is that when you access an array that is indexed by an enum's ordinal, it is your responsibility to use the correct `int` value; `int`s do not provide the type safety of enums. If you use the wrong value, the program will silently do the wrong thing or—if you're lucky—throw an `ArrayIndexOutOfBoundsException`.

There is a much better way to achieve the same effect. The array is effectively serving as a map from the enum to a value, so you might as well use a `Map`. More specifically, there is a very fast `Map` implementation designed for use with enum keys, known as `java.util.EnumMap`. Here is how the program looks when it is rewritten to use `EnumMap`:

```
// Using an EnumMap to associate data with an enum
Map<Plant.LifeCycle, Set<Plant>> plantsByLifeCycle =
    new EnumMap<>(Plant.LifeCycle.class);
for (Plant.LifeCycle lc : Plant.LifeCycle.values())
    plantsByLifeCycle.put(lc, new HashSet<>());
for (Plant p : garden)
    plantsByLifeCycle.get(p.lifeCycle).add(p);
System.out.println(plantsByLifeCycle);
```

This program is shorter, clearer, safer, and comparable in speed to the original version. There is no unsafe cast; no need to label the output manually because the map keys are enums that know how to translate themselves to printable strings; and no possibility for error in computing array indices. The reason that `EnumMap` is comparable in speed to an ordinal-indexed array is that `EnumMap` uses such an array internally, but it hides this implementation detail from the programmer, combining the richness and type safety of a `Map` with the speed of an array. Note that the `EnumMap` constructor takes the `Class` object of the key type: this is a *bounded type token*, which provides runtime generic type information (Item 33).

The previous program can be further shortened by using a stream (Item 45) to manage the map. Here is the simplest stream-based code that largely duplicates the behavior of the previous example:

```
// Naive stream-based approach - unlikely to produce an EnumMap!
System.out.println(Arrays.stream(garden)
        .collect(groupingBy(p -> p.lifeCycle)));
```

The problem with this code is that it chooses its own map implementation, and in practice it won't be an `EnumMap`, so it won't match the space and time performance of the version with the explicit `EnumMap`. To rectify this problem, use the three-

parameter form of `Collectors.groupingBy`, which allows the caller to specify the map implementation using the `mapFactory` parameter:

```
// Using a stream and an EnumMap to associate data with an enum
System.out.println(Arrays.stream(garden)
        .collect(groupingBy(p -> p.lifeCycle,
            () -> new EnumMap<>(LifeCycle.class), toSet())));
```

This optimization would not be worth doing in a toy program like this one but could be critical in a program that made heavy use of the map.

The behavior of the stream-based versions differs slightly from that of the `EmumMap` version. The `EnumMap` version always makes a nested map for each plant lifecycle, while the stream-based versions only make a nested map if the garden contains one or more plants with that lifecycle. So, for example, if the garden contains annuals and perennials but no biennials, the size of `plantsByLifeCycle` will be three in the `EnumMap` version and two in both of the stream-based versions.

You may see an array of arrays indexed (twice!) by ordinals used to represent a mapping from two enum values. For example, this program uses such an array to map two phases to a phase transition (liquid to solid is freezing, liquid to gas is boiling, and so forth):

```
// Using ordinal() to index array of arrays - DON'T DO THIS!
public enum Phase {
    SOLID, LIQUID, GAS;

    public enum Transition {
        MELT, FREEZE, BOIL, CONDENSE, SUBLIME, DEPOSIT;

        // Rows indexed by from-ordinal, cols by to-ordinal
        private static final Transition[][] TRANSITIONS = {
            { null,    MELT,     SUBLIME },
            { FREEZE,  null,     BOIL    },
            { DEPOSIT, CONDENSE, null    }
        };

        // Returns the phase transition from one phase to another
        public static Transition from(Phase from, Phase to) {
            return TRANSITIONS[from.ordinal()][to.ordinal()];
        }
    }
}
```

This program works and may even appear elegant, but appearances can be deceiving. Like the simpler garden example shown earlier, the compiler has no way of knowing the relationship between ordinals and array indices. If you make a

mistake in the transition table or forget to update it when you modify the `Phase` or `Phase.Transition` enum type, your program will fail at runtime. The failure may be an `ArrayIndexOutOfBoundsException`, a `NullPointerException`, or (worse) silent erroneous behavior. And the size of the table is quadratic in the number of phases, even if the number of non-null entries is smaller.

Again, you can do much better with `EnumMap`. Because each phase transition is indexed by a *pair* of phase enums, you are best off representing the relationship as a map from one enum (the "from" phase) to a map from the second enum (the "to" phase) to the result (the phase transition). The two phases associated with a phase transition are best captured by associating them with the phase transition enum, which can then be used to initialize the nested `EnumMap`:

```
// Using a nested EnumMap to associate data with enum pairs
public enum Phase {
   SOLID, LIQUID, GAS;

   public enum Transition {
      MELT(SOLID, LIQUID), FREEZE(LIQUID, SOLID),
      BOIL(LIQUID, GAS),   CONDENSE(GAS, LIQUID),
      SUBLIME(SOLID, GAS), DEPOSIT(GAS, SOLID);

      private final Phase from;
      private final Phase to;

      Transition(Phase from, Phase to) {
         this.from = from;
         this.to = to;
      }

      // Initialize the phase transition map
      private static final Map<Phase, Map<Phase, Transition>>
        m = Stream.of(values()).collect(groupingBy(t -> t.from,
         () -> new EnumMap<>(Phase.class),
         toMap(t -> t.to, t -> t,
            (x, y) -> y, () -> new EnumMap<>(Phase.class))));

      public static Transition from(Phase from, Phase to) {
         return m.get(from).get(to);
      }
   }
}
```

The code to initialize the phase transition map is a bit complicated. The type of the map is `Map<Phase, Map<Phase, Transition>>`, which means "map from (source) phase to map from (destination) phase to transition." This map-of-maps is initialized using a cascaded sequence of two collectors. The first collector groups

the transitions by source phase, and the second creates an `EnumMap` with mappings from destination phase to transition. The merge function in the second collector (`((x, y) -> y)`) is unused; it is required only because we need to specify a map factory in order to get an `EnumMap`, and `Collectors` provides telescoping factories. The previous edition of this book used explicit iteration to initialize the phase transition map. The code was more verbose but arguably easier to understand.

Now suppose you want to add a new phase to the system: *plasma*, or ionized gas. There are only two transitions associated with this phase: *ionization*, which takes a gas to a plasma; and *deionization*, which takes a plasma to a gas. To update the array-based program, you would have to add one new constant to `Phase` and two to `Phase.Transition`, and replace the original nine-element array of arrays with a new sixteen-element version. If you add too many or too few elements to the array or place an element out of order, you are out of luck: the program will compile, but it will fail at runtime. To update the `EnumMap`-based version, all you have to do is add `PLASMA` to the list of phases, and `IONIZE(GAS, PLASMA)` and `DEIONIZE(PLASMA, GAS)` to the list of phase transitions:

```
// Adding a new phase using the nested EnumMap implementation
public enum Phase {
    SOLID, LIQUID, GAS, PLASMA;

    public enum Transition {
        MELT(SOLID, LIQUID), FREEZE(LIQUID, SOLID),
        BOIL(LIQUID, GAS),   CONDENSE(GAS, LIQUID),
        SUBLIME(SOLID, GAS), DEPOSIT(GAS, SOLID),
        IONIZE(GAS, PLASMA), DEIONIZE(PLASMA, GAS);
        ... // Remainder unchanged
    }
}
```

The program takes care of everything else and leaves you virtually no opportunity for error. Internally, the map of maps is implemented with an array of arrays, so you pay little in space or time cost for the added clarity, safety, and ease of maintenance.

In the interest of brevity, the above examples use `null` to indicate the absence of a state change (wherein `to` and `from` are identical). This is not good practice and is likely to result in a `NullPointerException` at runtime. Designing a clean, elegant solution to this problem is surprisingly tricky, and the resulting programs are sufficiently long that they would detract from the primary material in this item.

In summary, **it is rarely appropriate to use ordinals to index into arrays: use `EnumMap` instead.** If the relationship you are representing is multidimensional, use `EnumMap<..., EnumMap<...>>`. This is a special case of the general principle that application programmers should rarely, if ever, use `Enum.ordinal` (Item 35).

Item 38: Emulate extensible enums with interfaces

In almost all respects, enum types are superior to the typesafe enum pattern described in the first edition of this book [Bloch01]. On the face of it, one exception concerns extensibility, which was possible under the original pattern but is not supported by the language construct. In other words, using the pattern, it was possible to have one enumerated type extend another; using the language feature, it is not. This is no accident. For the most part, extensibility of enums turns out to be a bad idea. It is confusing that elements of an extension type are instances of the base type and not vice versa. There is no good way to enumerate over all of the elements of a base type and its extensions. Finally, extensibility would complicate many aspects of the design and implementation.

That said, there is at least one compelling use case for extensible enumerated types, which is *operation codes,* also known as *opcodes.* An opcode is an enumerated type whose elements represent operations on some machine, such as the `Operation` type in Item 34, which represents the functions on a simple calculator. Sometimes it is desirable to let the users of an API provide their own operations, effectively extending the set of operations provided by the API.

Luckily, there is a nice way to achieve this effect using enum types. The basic idea is to take advantage of the fact that enum types can implement arbitrary interfaces by defining an interface for the opcode type and an enum that is the standard implementation of the interface. For example, here is an extensible version of the `Operation` type from Item 34:

```
// Emulated extensible enum using an interface
public interface Operation {
    double apply(double x, double y);
}

public enum BasicOperation implements Operation {
    PLUS("+") {
        public double apply(double x, double y) { return x + y; }
    },
    MINUS("-") {
        public double apply(double x, double y) { return x - y; }
    },
    TIMES("*") {
        public double apply(double x, double y) { return x * y; }
    },
    DIVIDE("/") {
        public double apply(double x, double y) { return x / y; }
    };
```

```
    private final String symbol;

    BasicOperation(String symbol) {
        this.symbol = symbol;
    }

    @Override public String toString() {
        return symbol;
    }
}
```

While the enum type (`BasicOperation`) is not extensible, the interface type (`Operation`) is, and it is the interface type that is used to represent operations in APIs. You can define another enum type that implements this interface and use instances of this new type in place of the base type. For example, suppose you want to define an extension to the operation type shown earlier, consisting of the exponentiation and remainder operations. All you have to do is write an enum type that implements the `Operation` interface:

```
// Emulated extension enum
public enum ExtendedOperation implements Operation {
    EXP("^") {
        public double apply(double x, double y) {
            return Math.pow(x, y);
        }
    },
    REMAINDER("%") {
        public double apply(double x, double y) {
            return x % y;
        }
    };

    private final String symbol;

    ExtendedOperation(String symbol) {
        this.symbol = symbol;
    }

    @Override public String toString() {
        return symbol;
    }
}
```

You can now use your new operations anywhere you could use the basic operations, provided that APIs are written to take the interface type (`Operation`), not the implementation (`BasicOperation`). Note that you don't have to declare the

abstract `apply` method in the enum as you do in a nonextensible enum with instance-specific method implementations (page 162). This is because the abstract method (`apply`) is a member of the interface (`Operation`).

Not only is it possible to pass a single instance of an "extension enum" anywhere a "base enum" is expected, but it is possible to pass in an entire extension enum type and use its elements in addition to or instead of those of the base type. For example, here is a version of the test program on page 163 that exercises all of the extended operations defined previously:

```
public static void main(String[] args) {
    double x = Double.parseDouble(args[0]);
    double y = Double.parseDouble(args[1]);
    test(ExtendedOperation.class, x, y);
}

private static <T extends Enum<T> & Operation> void test(
        Class<T> opEnumType, double x, double y) {
    for (Operation op : opEnumType.getEnumConstants())
        System.out.printf("%f %s %f = %f%n",
                          x, op, y, op.apply(x, y));
}
```

Note that the class literal for the extended operation type (`ExtendedOperation.class`) is passed from `main` to `test` to describe the set of extended operations. The class literal serves as a *bounded type token* (Item 33). The admittedly complex declaration for the `opEnumType` parameter (`<T extends Enum<T> & Operation> Class<T>`) ensures that the `Class` object represents both an enum and a subtype of `Operation`, which is exactly what is required to iterate over the elements and perform the operation associated with each one.

A second alternative is to pass a `Collection<? extends Operation>`, which is a *bounded wildcard type* (Item 31), instead of passing a class object:

```
public static void main(String[] args) {
    double x = Double.parseDouble(args[0]);
    double y = Double.parseDouble(args[1]);
    test(Arrays.asList(ExtendedOperation.values()), x, y);
}

private static void test(Collection<? extends Operation> opSet,
        double x, double y) {
    for (Operation op : opSet)
        System.out.printf("%f %s %f = %f%n",
                          x, op, y, op.apply(x, y));
}
```

The resulting code is a bit less complex, and the `test` method is a bit more flexible: it allows the caller to combine operations from multiple implementation types. On the other hand, you forgo the ability to use `EnumSet` (Item 36) and `EnumMap` (Item 37) on the specified operations.

Both programs shown previously will produce this output when run with command line arguments `4` and `2`:

```
4.000000 ^ 2.000000 = 16.000000
4.000000 % 2.000000 = 0.000000
```

A minor disadvantage of the use of interfaces to emulate extensible enums is that implementations cannot be inherited from one enum type to another. If the implementation code does not rely on any state, it can be placed in the interface, using default implementations (Item 20). In the case of our `Operation` example, the logic to store and retrieve the symbol associated with an operation must be duplicated in `BasicOperation` and `ExtendedOperation`. In this case it doesn't matter because very little code is duplicated. If there were a larger amount of shared functionality, you could encapsulate it in a helper class or a static helper method to eliminate the code duplication.

The pattern described in this item is used in the Java libraries. For example, the `java.nio.file.LinkOption` enum type implements the `CopyOption` and `OpenOption` interfaces.

In summary, **while you cannot write an extensible enum type, you can emulate it by writing an interface to accompany a basic enum type that implements the interface.** This allows clients to write their own enums (or other types) that implement the interface. Instances of these types can then be used wherever instances of the basic enum type can be used, assuming APIs are written in terms of the interface.

Item 39: Prefer annotations to naming patterns

Historically, it was common to use *naming patterns* to indicate that some program elements demanded special treatment by a tool or framework. For example, prior to release 4, the JUnit testing framework required its users to designate test methods by beginning their names with the characters `test` [Beck04]. This technique works, but it has several big disadvantages. First, typographical errors result in silent failures. For example, suppose you accidentally named a test method `tsetSafetyOverride` instead of `testSafetyOverride`. JUnit 3 wouldn't complain, but it wouldn't execute the test either, leading to a false sense of security.

A second disadvantage of naming patterns is that there is no way to ensure that they are used only on appropriate program elements. For example, suppose you called a class `TestSafetyMechanisms` in hopes that JUnit 3 would automatically test all of its methods, regardless of their names. Again, JUnit 3 wouldn't complain, but it wouldn't execute the tests either.

A third disadvantage of naming patterns is that they provide no good way to associate parameter values with program elements. For example, suppose you want to support a category of test that succeeds only if it throws a particular exception. The exception type is essentially a parameter of the test. You could encode the exception type name into the test method name using some elaborate naming pattern, but this would be ugly and fragile (Item 62). The compiler would have no way of knowing to check that the string that was supposed to name an exception actually did. If the named class didn't exist or wasn't an exception, you wouldn't find out until you tried to run the test.

Annotations [JLS, 9.7] solve all of these problems nicely, and JUnit adopted them starting with release 4. In this item, we'll write our own toy testing framework to show how annotations work. Suppose you want to define an annotation type to designate simple tests that are run automatically and fail if they throw an exception. Here's how such an annotation type, named `Test`, might look:

```
// Marker annotation type declaration
import java.lang.annotation.*;

/**
 * Indicates that the annotated method is a test method.
 * Use only on parameterless static methods.
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Test {
}
```

The declaration for the `Test` annotation type is itself annotated with `Retention` and `Target` annotations. Such annotations on annotation type declarations are known as *meta-annotations*. The `@Retention(RetentionPolicy.RUNTIME)` meta-annotation indicates that `Test` annotations should be retained at runtime. Without it, `Test` annotations would be invisible to the test tool. The `@Target.get(ElementType.METHOD)` meta-annotation indicates that the `Test` annotation is legal only on method declarations: it cannot be applied to class declarations, field declarations, or other program elements.

The comment before the `Test` annotation declaration says, "Use only on parameterless static methods." It would be nice if the compiler could enforce this, but it can't, unless you write an *annotation processor* to do so. For more on this topic, see the documentation for `javax.annotation.processing`. In the absence of such an annotation processor, if you put a `Test` annotation on the declaration of an instance method or on a method with one or more parameters, the test program will still compile, leaving it to the testing tool to deal with the problem at runtime.

Here is how the `Test` annotation looks in practice. It is called a *marker annotation* because it has no parameters but simply "marks" the annotated element. If the programmer were to misspell `Test` or to apply the `Test` annotation to a program element other than a method declaration, the program wouldn't compile:

```
// Program containing marker annotations
public class Sample {
    @Test public static void m1() { }  // Test should pass
    public static void m2() { }
    @Test public static void m3() {    // Test should fail
        throw new RuntimeException("Boom");
    }
    public static void m4() { }
    @Test public void m5() { } // INVALID USE: nonstatic method
    public static void m6() { }
    @Test public static void m7() {    // Test should fail
        throw new RuntimeException("Crash");
    }
    public static void m8() { }
}
```

The `Sample` class has seven static methods, four of which are annotated as tests. Two of these, `m3` and `m7`, throw exceptions, and two, `m1` and `m5`, do not. But one of the annotated methods that does not throw an exception, `m5`, is an instance method, so it is not a valid use of the annotation. In sum, `Sample` contains four tests: one will pass, two will fail, and one is invalid. The four methods that are not annotated with the `Test` annotation will be ignored by the testing tool.

The `Test` annotations have no direct effect on the semantics of the `Sample` class. They serve only to provide information for use by interested programs. More generally, annotations don't change the semantics of the annotated code but enable it for special treatment by tools such as this simple test runner:

```
// Program to process marker annotations
import java.lang.reflect.*;

public class RunTests {
    public static void main(String[] args) throws Exception {
        int tests = 0;
        int passed = 0;
        Class<?> testClass = Class.forName(args[0]);
        for (Method m : testClass.getDeclaredMethods()) {
            if (m.isAnnotationPresent(Test.class)) {
                tests++;
                try {
                    m.invoke(null);
                    passed++;
                } catch (InvocationTargetException wrappedExc) {
                    Throwable exc = wrappedExc.getCause();
                    System.out.println(m + " failed: " + exc);
                } catch (Exception exc) {
                    System.out.println("Invalid @Test: " + m);
                }
            }
        }
        System.out.printf("Passed: %d, Failed: %d%n",
                          passed, tests - passed);
    }
}
```

The test runner tool takes a fully qualified class name on the command line and runs all of the class's `Test`-annotated methods reflectively, by calling `Method.invoke`. The `isAnnotationPresent` method tells the tool which methods to run. If a test method throws an exception, the reflection facility wraps it in an `InvocationTargetException`. The tool catches this exception and prints a failure report containing the original exception thrown by the test method, which is extracted from the `InvocationTargetException` with the `getCause` method.

If an attempt to invoke a test method by reflection throws any exception other than `InvocationTargetException`, it indicates an invalid use of the `Test` annotation that was not caught at compile time. Such uses include annotation of an instance method, of a method with one or more parameters, or of an inaccessible method. The second catch block in the test runner catches these `Test` usage errors

and prints an appropriate error message. Here is the output that is printed if `RunTests` is run on `Sample`:

```
public static void Sample.m3() failed: RuntimeException: Boom
Invalid @Test: public void Sample.m5()
public static void Sample.m7() failed: RuntimeException: Crash
Passed: 1, Failed: 3
```

Now let's add support for tests that succeed only if they throw a particular exception. We'll need a new annotation type for this:

```
// Annotation type with a parameter
import java.lang.annotation.*;
/**
 * Indicates that the annotated method is a test method that
 * must throw the designated exception to succeed.
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTest {
    Class<? extends Throwable> value();
}
```

The type of the parameter for this annotation is `Class<? extends Throwable>`. This wildcard type is, admittedly, a mouthful. In English, it means "the `Class` object for some class that extends `Throwable`," and it allows the user of the annotation to specify any exception (or error) type. This usage is an example of a *bounded type token* (Item 33). Here's how the annotation looks in practice. Note that class literals are used as the values for the annotation parameter:

```
// Program containing annotations with a parameter
public class Sample2 {
    @ExceptionTest(ArithmeticException.class)
    public static void m1() {  // Test should pass
        int i = 0;
        i = i / i;
    }
    @ExceptionTest(ArithmeticException.class)
    public static void m2() {  // Should fail (wrong exception)
        int[] a = new int[0];
        int i = a[1];
    }
    @ExceptionTest(ArithmeticException.class)
    public static void m3() { }  // Should fail (no exception)
}
```

Now let's modify the test runner tool to process the new annotation. Doing so consists of adding the following code to the `main` method:

```
if (m.isAnnotationPresent(ExceptionTest.class)) {
    tests++;
    try {
        m.invoke(null);
        System.out.printf("Test %s failed: no exception%n", m);
    } catch (InvocationTargetException wrappedEx) {
        Throwable exc = wrappedEx.getCause();
        Class<? extends Throwable> excType =
            m.getAnnotation(ExceptionTest.class).value();
        if (excType.isInstance(exc)) {
            passed++;
        } else {
            System.out.printf(
                "Test %s failed: expected %s, got %s%n",
                m, excType.getName(), exc);
        }
    } catch (Exception exc) {
        System.out.println("Invalid @Test: " + m);
    }
}
```

This code is similar to the code we used to process `Test` annotations, with one exception: this code extracts the value of the annotation parameter and uses it to check if the exception thrown by the test is of the right type. There are no explicit casts, and hence no danger of a `ClassCastException`. The fact that the test program compiled guarantees that its annotation parameters represent valid exception types, with one caveat: if the annotation parameters were valid at compile time but the class file representing a specified exception type is no longer present at runtime, the test runner will throw `TypeNotPresentException`.

Taking our exception testing example one step further, it is possible to envision a test that passes if it throws any one of several specified exceptions. The annotation mechanism has a facility that makes it easy to support this usage. Suppose we change the parameter type of the `ExceptionTest` annotation to be an array of `Class` objects:

```
// Annotation type with an array parameter
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTest {
    Class<? extends Exception>[] value();
}
```

The syntax for array parameters in annotations is flexible. It is optimized for single-element arrays. All of the previous `ExceptionTest` annotations are still valid with the new array-parameter version of `ExceptionTest` and result in single-element arrays. To specify a multiple-element array, surround the elements with curly braces and separate them with commas:

```
// Code containing an annotation with an array parameter
@ExceptionTest({ IndexOutOfBoundsException.class,
                 NullPointerException.class })
public static void doublyBad() {
    List<String> list = new ArrayList<>();

    // The spec permits this method to throw either
    // IndexOutOfBoundsException or NullPointerException
    list.addAll(5, null);
}
```

It is reasonably straightforward to modify the test runner tool to process the new version of `ExceptionTest`. This code replaces the original version:

```
if (m.isAnnotationPresent(ExceptionTest.class)) {
    tests++;
    try {
        m.invoke(null);
        System.out.printf("Test %s failed: no exception%n", m);
    } catch (Throwable wrappedExc) {
        Throwable exc = wrappedExc.getCause();
        int oldPassed = passed;
        Class<? extends Exception>[] excTypes =
            m.getAnnotation(ExceptionTest.class).value();
        for (Class<? extends Exception> excType : excTypes) {
            if (excType.isInstance(exc)) {
                passed++;
                break;
            }
        }
        if (passed == oldPassed)
            System.out.printf("Test %s failed: %s %n", m, exc);
    }
}
```

As of Java 8, there is another way to do multivalued annotations. Instead of declaring an annotation type with an array parameter, you can annotate the declaration of an annotation with the `@Repeatable` meta-annotation, to indicate that the annotation may be applied repeatedly to a single element. This meta-annotation

takes a single parameter, which is the class object of a *containing annotation type*, whose sole parameter is an array of the annotation type [JLS, 9.6.3]. Here's how the annotation declarations look if we take this approach with our `ExceptionTest` annotation. Note that the containing annotation type must be annotated with an appropriate retention policy and target, or the declarations won't compile:

```
// Repeatable annotation type
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@Repeatable(ExceptionTestContainer.class)
public @interface ExceptionTest {
    Class<? extends Exception> value();
}

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTestContainer {
    ExceptionTest[] value();
}
```

Here's how our `doublyBad` test looks with a repeated annotation in place of an array-valued annotation:

```
// Code containing a repeated annotation
@ExceptionTest(IndexOutOfBoundsException.class)
@ExceptionTest(NullPointerException.class)
public static void doublyBad() { ... }
```

Processing repeatable annotations requires care. A repeated annotation generates a synthetic annotation of the containing annotation type. The `getAnnotationsByType` method glosses over this fact, and can be used to access both repeated and non-repeated annotations of a repeatable annotation type. But `isAnnotationPresent` makes it explicit that repeated annotations are not of the annotation type, but of the containing annotation type. If an element has a repeated annotation of some type and you use the `isAnnotationPresent` method to check if the element has an annotation of that type, you'll find that it does not. Using this method to check for the presence of an annotation type will therefore cause your program to silently ignore repeated annotations. Similarly, using this method to check for the containing annotation type will cause the program to silently ignore non-repeated annotations. To detect repeated and non-repeated annotations with `isAnnotationPresent`, you much check for both the annotation type and its containing annotation type. Here's how the relevant part of our

`RunTests` program looks when modified to use the repeatable version of the `ExceptionTest` annotation:

```
// Processing repeatable annotations
if (m.isAnnotationPresent(ExceptionTest.class)
    || m.isAnnotationPresent(ExceptionTestContainer.class)) {
    tests++;
    try {
        m.invoke(null);
        System.out.printf("Test %s failed: no exception%n", m);
    } catch (Throwable wrappedExc) {
        Throwable exc = wrappedExc.getCause();
        int oldPassed = passed;
        ExceptionTest[] excTests =
                m.getAnnotationsByType(ExceptionTest.class);
        for (ExceptionTest excTest : excTests) {
            if (excTest.value().isInstance(exc)) {
                passed++;
                break;
            }
        }
        if (passed == oldPassed)
            System.out.printf("Test %s failed: %s %n", m, exc);
    }
}
```

Repeatable annotations were added to improve the readability of source code that logically applies multiple instances of the same annotation type to a given program element. If you feel they enhance the readability of your source code, use them, but remember that there is more boilerplate in declaring and processing repeatable annotations, and that processing repeatable annotations is error-prone.

The testing framework in this item is just a toy, but it clearly demonstrates the superiority of annotations over naming patterns, and it only scratches the surface of what you can do with them. If you write a tool that requires programmers to add information to source code, define appropriate annotation types. **There is simply no reason to use naming patterns when you can use annotations instead.**

That said, with the exception of toolsmiths, most programmers will have no need to define annotation types. But **all programmers should use the predefined annotation types that Java provides** (Items 40, 27). Also, consider using the annotations provided by your IDE or static analysis tools. Such annotations can improve the quality of the diagnostic information provided by these tools. Note, however, that these annotations have yet to be standardized, so you may have some work to do if you switch tools or if a standard emerges.

Item 40: Consistently use the `Override` annotation

The Java libraries contain several annotation types. For the typical programmer, the most important of these is `@Override`. This annotation can be used only on method declarations, and it indicates that the annotated method declaration overrides a declaration in a supertype. If you consistently use this annotation, it will protect you from a large class of nefarious bugs. Consider this program, in which the class `Bigram` represents a *bigram*, or ordered pair of letters:

```
// Can you spot the bug?
public class Bigram {
    private final char first;
    private final char second;

    public Bigram(char first, char second) {
        this.first  = first;
        this.second = second;
    }
    public boolean equals(Bigram b) {
        return b.first == first && b.second == second;
    }
    public int hashCode() {
        return 31 * first + second;
    }

    public static void main(String[] args) {
        Set<Bigram> s = new HashSet<>();
        for (int i = 0; i < 10; i++)
            for (char ch = 'a'; ch <= 'z'; ch++)
                s.add(new Bigram(ch, ch));
        System.out.println(s.size());
    }
}
```

The main program repeatedly adds twenty-six bigrams, each consisting of two identical lowercase letters, to a set. Then it prints the size of the set. You might expect the program to print `26`, as sets cannot contain duplicates. If you try running the program, you'll find that it prints not `26` but `260`. What is wrong with it?

Clearly, the author of the `Bigram` class intended to override the `equals` method (Item 10) and even remembered to override `hashCode` in tandem (Item 11). Unfortunately, our hapless programmer failed to override `equals`, overloading it instead (Item 52). To override `Object.equals`, you must define an `equals` method whose parameter is of type `Object`, but the parameter of `Bigram`'s

`equals` method is not of type `Object`, so `Bigram` inherits the `equals` method from `Object`. This `equals` method tests for object *identity*, just like the `==` operator. Each of the ten copies of each bigram is distinct from the other nine, so they are deemed unequal by `Object.equals`, which explains why the program prints `260`.

Luckily, the compiler can help you find this error, but only if you help it by telling it that you intend to override `Object.equals`. To do this, annotate `Bigram.equals` with `@Override`, as shown here:

```
@Override public boolean equals(Bigram b) {
    return b.first == first && b.second == second;
}
```

If you insert this annotation and try to recompile the program, the compiler will generate an error message like this:

```
Bigram.java:10: method does not override or implement a method
from a supertype
    @Override public boolean equals(Bigram b) {
    ^
```

You will immediately realize what you did wrong, slap yourself on the forehead, and replace the broken `equals` implementation with a correct one (Item 10):

```
@Override public boolean equals(Object o) {
    if (!(o instanceof Bigram))
        return false;
    Bigram b = (Bigram) o;
    return b.first == first && b.second == second;
}
```

Therefore, you should **use the `Override` annotation on every method declaration that you believe to override a superclass declaration.** There is one minor exception to this rule. If you are writing a class that is not labeled abstract and you believe that it overrides an abstract method in its superclass, you needn't bother putting the `Override` annotation on that method. In a class that is not declared abstract, the compiler will emit an error message if you fail to override an abstract superclass method. However, you might wish to draw attention to all of the methods in your class that override superclass methods, in which case you should feel free to annotate these methods too. Most IDEs can be set to insert `Override` annotations automatically when you elect to override a method.

Most IDEs provide another reason to use the `Override` annotation consistently. If you enable the appropriate check, the IDE will generate a warning if you have a method that doesn't have an `Override` annotation but does override a superclass method. If you use the `Override` annotation consistently, these warnings will alert you to unintentional overriding. They complement the compiler's error messages, which alert you to unintentional failure to override. Between the IDE and the compiler, you can be sure that you're overriding methods everywhere you want to and nowhere else.

The `Override` annotation may be used on method declarations that override declarations from interfaces as well as classes. With the advent of default methods, it is good practice to use `Override` on concrete implementations of interface methods to ensure that the signature is correct. If you know that an interface does not have default methods, you may choose to omit `Override` annotations on concrete implementations of interface methods to reduce clutter.

In an abstract class or an interface, however, it *is* worth annotating *all* methods that you believe to override superclass or superinterface methods, whether concrete or abstract. For example, the `Set` interface adds no new methods to the `Collection` interface, so it should include `Override` annotations on all of its method declarations to ensure that it does not accidentally add any new methods to the `Collection` interface.

In summary, the compiler can protect you from a great many errors if you use the `Override` annotation on every method declaration that you believe to override a supertype declaration, with one exception. In concrete classes, you need not annotate methods that you believe to override abstract method declarations (though it is not harmful to do so).

Item 41: Use marker interfaces to define types

A *marker interface* is an interface that contains no method declarations but merely designates (or "marks") a class that implements the interface as having some property. For example, consider the `Serializable` interface (Chapter 12). By implementing this interface, a class indicates that its instances can be written to an `ObjectOutputStream` (or "serialized").

You may hear it said that marker annotations (Item 39) make marker interfaces obsolete. This assertion is incorrect. Marker interfaces have two advantages over marker annotations. First and foremost, **marker interfaces define a type that is implemented by instances of the marked class; marker annotations do not.** The existence of a marker interface type allows you to catch errors at compile time that you couldn't catch until runtime if you used a marker annotation.

Java's serialization facility (Chapter 6) uses the `Serializable` marker interface to indicate that a type is serializable. The `ObjectOutputStream.writeObject` method, which serializes the object that is passed to it, requires that its argument be serializable. Had the argument of this method been of type `Serializable`, an attempt to serialize an inappropriate object would have been detected at compile time (by type checking). Compile-time error detection is the intent of marker interfaces, but unfortunately, the `ObjectOutputStream.write` API does not take advantage of the `Serializable` interface: its argument is declared to be of type `Object`, so attempts to serialize an unserializable object won't fail until runtime.

Another advantage of marker interfaces over marker annotations is that they can be targeted more precisely. If an annotation type is declared with target `ElementType.TYPE`, it can be applied to *any* class or interface. Suppose you have a marker that is applicable only to implementations of a particular interface. If you define it as a marker interface, you can have it extend the sole interface to which it is applicable, guaranteeing that all marked types are also subtypes of the sole interface to which it is applicable.

Arguably, the `Set` interface is just such a *restricted marker interface*. It is applicable only to `Collection` subtypes, but it adds no methods beyond those defined by `Collection`. It is not generally considered to be a marker interface because it refines the contracts of several `Collection` methods, including `add`, `equals`, and `hashCode`. But it is easy to imagine a marker interface that is applicable only to subtypes of some particular interface and does *not* refine the contracts of any of the interface's methods. Such a marker interface might describe some invariant of the entire object or indicate that instances are eligible for processing

by a method of some other class (in the way that the `Serializable` interface indicates that instances are eligible for processing by `ObjectOutputStream`).

The chief advantage of marker annotations over marker interfaces is that they are part of the larger annotation facility. Therefore, marker annotations allow for consistency in annotation-based frameworks.

So when should you use a marker annotation and when should you use a marker interface? Clearly you must use an annotation if the marker applies to any program element other than a class or interface, because only classes and interfaces can be made to implement or extend an interface. If the marker applies only to classes and interfaces, ask yourself the question "Might I want to write one or more methods that accept only objects that have this marking?" If so, you should use a marker interface in preference to an annotation. This will make it possible for you to use the interface as a parameter type for the methods in question, which will result in the benefit of compile-time type checking. If you can convince yourself that you'll never want to write a method that accepts only objects with the marking, then you're probably better off using a marker annotation. If, additionally, the marking is part of a framework that makes heavy use of annotations, then a marker annotation is the clear choice.

In summary, marker interfaces and marker annotations both have their uses. If you want to define a type that does not have any new methods associated with it, a marker interface is the way to go. If you want to mark program elements other than classes and interfaces or to fit the marker into a framework that already makes heavy use of annotation types, then a marker annotation is the correct choice. **If you find yourself writing a marker annotation type whose target is `ElementType.TYPE`, take the time to figure out whether it really should be an annotation type or whether a marker interface would be more appropriate.**

In a sense, this item is the inverse of Item 22, which says, "If you don't want to define a type, don't use an interface." To a first approximation, this item says, "If you do want to define a type, do use an interface."