CHAPTER 5

Substitutes for C Constructs

THE Java programming language shares many similarities with the C programming language, but several C constructs have been omitted. In most cases, it's obvious why a C construct was omitted and how to make do without it. This chapter suggests replacements for several omitted C constructs whose replacements are not so obvious.

The common thread that connects the items in this chapter is that all of the omitted constructs are data-oriented rather than object-oriented. The Java programming language provides a powerful type system, and the suggested replacements take full advantage of that type system to deliver a higher quality abstraction than the C constructs they replace.

Even if you choose to skip this chapter, it's probably worth reading Item 21, which discusses the *typesafe enum* pattern, a replacement for C's `enum` construct. This pattern is not widely known at the time of this writing, and it has several advantages over the methods currently in common use.

Item 19: Replace structures with classes

The C `struct` construct was omitted from the Java programming language because a class does everything a structure does and more. A structure merely groups multiple data fields into a single object; a class associates operations with the resulting object and allows the data fields to be hidden from users of the object. In other words, a class can *encapsulate* its data into an object that is accessed solely by its methods, allowing the implementor the freedom to change the representation over time (Item 12).

Upon first exposure to the Java programming language, some C programmers believe that classes are too heavyweight to replace structures under some circum-

stances, but this is not the case. Degenerate classes consisting solely of data fields are loosely equivalent to C structures:

```
// Degenerate classes like this should not be public!
class Point {
    public float x;
    public float y;
}
```

Because such classes are accessed by their data fields, they do not offer the benefits of encapsulation. You cannot change the representation of such a class without changing its API, you cannot enforce any invariants, and you cannot take any auxiliary action when a field is modified. Hard-line object-oriented programmers feel that such classes are anathema and should always be replaced by classes with private fields and public *accessor methods*:

```
// Encapsulated structure class
class Point {
    private float x;
    private float y;

    public Point(float x, float y) {
        this.x = x;
        this.y = y;
    }

    public float getX() { return x; }
    public float getY() { return y; }

    public void setX(float x) { this.x = x; }
    public void setY(float y) { this.y = y; }
}
```

Certainly, the hard-liners are correct when it comes to public classes: If a class is accessible outside the confines of its package, the prudent programmer will provide accessor methods to preserve the flexibility to change the class's internal representation. If a public class were to expose its data fields, all hope of changing the representation would be lost, as client code for public classes can be distributed all over the known universe.

If, however, a class is package-private, or it is a private nested class, there is nothing inherently wrong with directly exposing its data fields—assuming they really do describe the abstraction provided by the class. This approach generates less visual clutter than the access method approach, both in the class definition

and in the client code that uses the class. While the client code is tied to the internal representation of the class, this code is restricted to the package that contains the class. In the unlikely event that a change in representation becomes desirable, it is possible to effect the change without touching any code outside the package. In the case of a private nested class, the scope of the change is further restricted to the enclosing class.

Several classes in the Java platform libraries violate the advice that public classes should not expose fields directly. Prominent examples include the `Point` and `Dimension` classes in the `java.awt` package. Rather than examples to be emulated, these classes should be regarded as cautionary tales. As described in Item 37, the decision to expose the internals of the Dimension class resulted in a serious performance problem that could not be solved without affecting clients.

Item 20: Replace unions with class hierarchies

The C `union` construct is most frequently used to define structures capable of holding more than one type of data. Such a structure typically contains at least two fields: a union and a *tag*. The tag is just an ordinary field used to indicate which of the possible types is held by the union. The tag is generally of some `enum` type. A structure containing a union and a tag is sometimes called a *discriminated union*.

In the C example below, the `shape_t` type is a discriminated union that can be used to represent either a rectangle or a circle. The `area` function takes a pointer to a `shape_t` structure and returns its area, or `-1.0`, if the structure is invalid:

```
/* Discriminated union */
#include "math.h"
typedef enum {RECTANGLE, CIRCLE} shapeType_t;

typedef struct {
    double length;
    double width;
} rectangleDimensions_t;

typedef struct {
    double radius;
} circleDimensions_t;

typedef struct {
    shapeType_t tag;
    union {
        rectangleDimensions_t rectangle;
        circleDimensions_t    circle;
    } dimensions;
} shape_t;

double area(shape_t *shape) {
    switch(shape->tag) {
      case RECTANGLE: {
        double length = shape->dimensions.rectangle.length;
        double width  = shape->dimensions.rectangle.width;
        return length * width;
      }
      case CIRCLE: {
        double r = shape->dimensions.circle.radius;
        return M_PI * (r*r);
      }
      default: return -1.0; /* Invalid tag */
    }
}
```

The designers of the Java programming language chose to omit the `union` construct because there is a much better mechanism for defining a single data type capable of representing objects of various types: subtyping. A discriminated union is really just a pallid imitation of a class hierarchy.

To transform a discriminated union into a class hierarchy, define an abstract class containing an abstract method for each operation whose behavior depends on the value of the tag. In the earlier example, there is only one such operation, area. This abstract class is the root of the class hierarchy. If there are any operations whose behavior does not depend on the value of the tag, turn these operations into concrete methods in the root class. Similarly, if there are any data fields in the discriminated union besides the tag and the union, these fields represent data common to all types and should be added to the root class. There are no such type-independent operations or data fields in the example.

Next, define a concrete subclass of the root class for each type that can be represented by the discriminated union. In the earlier example, the types are circle and rectangle. Include in each subclass the data fields particular to its type. In the example, radius is particular to circle, and length and width are particular to rectangle. Also include in each subclass the appropriate implementation of each abstract method in the root class. Here is the class hierarchy corresponding to the discriminated union example:

```
abstract class Shape {
    abstract double area();
}

class Circle extends Shape {
    final double radius;

    Circle(double radius) { this.radius = radius; }

    double area() { return Math.PI * radius*radius; }
}

class Rectangle extends Shape {
    final double length;
    final double width;

    Rectangle(double length, double width) {
        this.length = length;
        this.width  = width;
    }
    double area() { return length * width; }
}
```

A class hierarchy has numerous advantages over a discriminated union. Chief among these is that the class hierarchy provides type safety. In the example, every `Shape` instance is either a valid `Circle` or a valid `Rectangle`. It is a simple matter to generate a `shape_t` structure that is complete garbage, as the association between the tag and the union is not enforced by the language. If the tag indicates that the `shape_t` represents a rectangle but the union has been set for a circle, all bets are off. Even if a discriminated union has been initialized properly, it is possible to pass it to a function that is inappropriate for its tag value.

A second advantage of the class hierarchy is that code is simple and clear. The discriminated union is cluttered with boilerplate: declaring the `enum` type, declaring the tag field, switching on the tag field, dealing with unexpected tag values, and the like. The discriminated union code is made even less readable by the fact that the operations for the various types are intermingled rather than segregated by type.

A third advantage of the class hierarchy is that it is easily extensible, even by multiple parties working independently. To extend a class hierarchy, simply add a new subclass. If you forget to override one of the abstract methods in the superclass, the compiler will tell you in no uncertain terms. To extend a discriminated union, you need access to the source code. You must add a new value to the `enum` type, as well as a new case to the `switch` statement in each operation on the discriminated union. Finally, you must recompile. If you forget to provide a new case for some method, you won't find out until run time, and then only if you're careful to check for unrecognized tag values and generate an appropriate error message.

A fourth advantage of the class hierarchy is that it can be made to reflect natural hierarchical relationships among types, to allow for increased flexibility and better compile-time type checking. Suppose the discriminated union in the original example also allowed for squares. The class hierarchy could be made to reflect the fact a square is a special kind of rectangle (assuming both are immutable):

```
class Square extends Rectangle {
    Square(double side) {
        super(side, side);
    }

    double side() {
        return length; // or equivalently, width
    }
}
```

The class hierarchy in this example is not the only one that could have been written to replace the discriminated union. The hierarchy embodies several design decisions worthy of note. The classes in the hierarchy, with the exception of `Square`, are accessed by their fields rather than by accessor methods. This was done for brevity and would be unacceptable if the classes were public (Item 19). The classes are immutable, which is not always appropriate, but is generally a good thing (Item 13).

Since the Java programming language does not provide the `union` construct, you might think there's no danger of implementing a discriminated union, but it is possible to write code with many of the same disadvantages. Whenever you're tempted to write a class with an explicit tag field, think about whether the tag could be eliminated and the class replaced by a class hierarchy.

Another use of C's `union` construct, completely unrelated to discriminated unions, involves looking at the internal representation of a piece of data, intentionally violating the type system. This usage is demonstrated by the following C code fragment, which prints the machine-specific hex representation of a `float`:

```
union {
    float f;
    int   bits;
} sleaze;

sleaze.f = 6.699e-41;    /* Put data in one field of union... */
printf("%x\n", sleaze.bits); /* ...and read it out the other. */
```

While it can be useful, especially for system programming, this nonportable usage has no counterpart in the Java programming language. In fact, it is antithetical to the spirit of the language, which guarantees type safety and goes to great lengths to insulate programmers from machine-specific internal representations.

The `java.lang` package does contain methods to translate floating point numbers into bit representations, but these methods are defined in terms of a precisely specified bit representation to ensure portability. The code fragment that follows, which is loosely equivalent to the earlier C fragment, is guaranteed to print the same result, no matter where it's run:

```
System.out.println(
    Integer.toHexString(Float.floatToIntBits(6.699e-41f)));
```

Item 21: Replace enum constructs with classes

The C enum construct was omitted from the Java programming language. Nominally, this construct defines an *enumerated type*: a type whose legal values consist of a fixed set of constants. Unfortunately, the enum construct doesn't do a very good job of defining enumerated types. It just defines a set of named integer constants, providing nothing in the way of type safety and little in the way of convenience. Not only is the following legal C:

```
typedef enum {FUJI, PIPPIN, GRANNY_SMITH} apple_t;
typedef enum {NAVEL, TEMPLE, BLOOD} orange_t;
orange_t myFavorite = PIPPIN;    /* Mixing apples and oranges */
```

but so is this atrocity:

```
orange_t x = (FUJI - PIPPIN)/TEMPLE;    /* Applesauce! */
```

The enum construct does not establish a name space for the constants it generates. Therefore the following declaration, which reuses one of the names, conflicts with the orange_t declaration:

```
typedef enum {BLOOD, SWEAT, TEARS} fluid_t;
```

Types defined with the enum construct are brittle. Adding constants to such a type without recompiling its clients causes unpredictable behavior, unless care is taken to preserve all of the preexisting constant values. Multiple parties cannot add constants to such a type independently, as their new enumeration constants are likely to conflict. The enum construct provides no easy way to translate enumeration constants into printable strings or to enumerate over the constants in a type.

Unfortunately, the most commonly used pattern for enumerated types in the Java programming language, shown here, shares the shortcomings of the C enum construct:

```
// The int enum pattern - problematic!!
public class PlayingCard {
    public static final int SUIT_CLUBS    = 0;
    public static final int SUIT_DIAMONDS = 1;
    public static final int SUIT_HEARTS   = 2;
    public static final int SUIT_SPADES   = 3;
    ...
}
```

You may encounter a variant of this pattern in which `String` constants are used in place of `int` constants. This variant should never be used. While it does provide printable strings for its constants, it can lead to performance problems because it relies on string comparisons. Furthermore, it can lead naive users to hard-code string constants into client code instead of using the appropriate field names. If such a hard-coded string constant contains a typographical error, the error will escape detection at compile time and result in bugs at run time.

Luckily, the Java programming language presents an alternative that avoids all the shortcomings of the common `int` and `String` patterns and provides many added benefits. It is called the *typesafe enum* pattern. Unfortunately, it is not yet widely known. The basic idea is simple: Define a class representing a single element of the enumerated type, and don't provide any public constructors. Instead, provide public static final fields, one for each constant in the enumerated type. Here's how the pattern looks in its simplest form:

```
// The typesafe enum pattern
public class Suit {
    private final String name;

    private Suit(String name) { this.name = name; }

    public String toString()  { return name; }

    public static final Suit CLUBS    = new Suit("clubs");
    public static final Suit DIAMONDS = new Suit("diamonds");
    public static final Suit HEARTS   = new Suit("hearts");
    public static final Suit SPADES   = new Suit("spades");
}
```

Because there is no way for clients to create objects of the class or to extend it, there will never be any objects of the type besides those exported via the public static final fields. Even though the class is not declared final, there is no way to extend it: Subclass constructors must invoke a superclass constructor, and no such constructor is accessible.

As its name implies, the typesafe enum pattern provides compile-time type safety. If you declare a method with a parameter of type `Suit`, you are guaranteed that any non-null object reference passed in represents one of the four valid suits. Any attempt to pass an incorrectly typed object will be caught at compile time, as will any attempt to assign an expression of one enumerated type to a variable of another. Multiple typesafe enum classes with identically named enumeration constants coexist peacefully because each class has its own name space.

Constants may be added to a typesafe enum class without recompiling its clients because the public static object reference fields containing the enumeration constants provide a layer of insulation between the client and the enum class. The constants themselves are never compiled into clients as they are in the more common `int` pattern and its `String` variant.

Because typesafe enums are full-fledged classes, you can override the `toString` method as shown earlier, allowing values to be translated into printable strings. You can, if you desire, go one step further and internationalize typesafe enums by standard means. Note that string names are used only by the `toString` method; they are not used for equality comparisons, as the `equals` implementation, which is inherited from `Object`, performs a reference identity comparison.

More generally, you can augment a typesafe enum class with any method that seems appropriate. Our `Suit` class, for example, might benefit from the addition of a method that returns the color of the suit or one that returns an image representing the suit. A class can start life as a simple typesafe enum and evolve over time into a full-featured abstraction.

Because arbitrary methods can be added to typesafe enum classes, they can be made to implement any interface. For example, suppose that you want `Suit` to implement `Comparable` so clients can sort bridge hands by suit. Here's a slight variant on the original pattern that accomplishes this feat. A static variable, `nextOrdinal`, is used to assign an ordinal number to each instance as it is created. These ordinals are used by the `compareTo` method to order instances:

```
// Ordinal-based typesafe enum
public class Suit implements Comparable {
    private final String name;

    // Ordinal of next suit to be created
    private static int nextOrdinal = 0;

    // Assign an ordinal to this suit
    private final int ordinal = nextOrdinal++;

    private Suit(String name) { this.name = name; }

    public String toString()  { return name; }

    public int compareTo(Object o) {
        return ordinal - ((Suit)o).ordinal;
    }
```

```
    public static final Suit CLUBS    = new Suit("clubs");
    public static final Suit DIAMONDS = new Suit("diamonds");
    public static final Suit HEARTS   = new Suit("hearts");
    public static final Suit SPADES   = new Suit("spades");
}
```

Because typesafe enum constants are objects, you can put them into collections. For example, suppose you want the `Suit` class to export an immutable list of the suits in standard order. Merely add these two field declarations to the class:

```
private static final Suit[] PRIVATE_VALUES =
   { CLUBS, DIAMONDS, HEARTS, SPADES };
public static final List VALUES =
   Collections.unmodifiableList(Arrays.asList(PRIVATE_VALUES));
```

Unlike the simplest form of the typesafe enum pattern, classes of the ordinal-based form above can be made serializable (Chapter 10) with a little care. It is not sufficient merely to add `implements Serializable` to the class declaration. You must also provide a `readResolve` method (Item 57):

```
private Object readResolve() throws ObjectStreamException {
    return PRIVATE_VALUES[ordinal]; // Canonicalize
}
```

This method, which is invoked automatically by the serialization system, prevents duplicate constants from coexisting as a result of deserialization. This maintains the guarantee that only a single object represents each enum constant, avoiding the need to override `Object.equals`. Without this guarantee, `Object.equals` would report a false negative when presented with two equal but distinct enumeration constants. Note that the `readResolve` method refers to the `PRIVATE_VALUES` array, so you must declare this array even if you choose not to export `VALUES`. Note also that the `name` field is not used by the `readResolve` method, so it can and should be made transient.

The resulting class is somewhat brittle; constructors for any new values must appear after those of all existing values, to ensure that previously serialized instances do not change their value when they're deserialized. This is so because the serialized form (Item 55) of an enumeration constant consists solely of its ordinal. If the enumeration constant pertaining to an ordinal changes, a serialized constant with that ordinal will take on the new value when it is deserialized.

There may be one or more pieces of behavior associated with each constant that are used only from within the package containing the typesafe enum class.

Such behaviors are best implemented as package-private methods on the class. Each enum constant then carries with it a hidden collection of behaviors that allows the package containing the enumerated type to react appropriately when presented with the constant.

If a typesafe enum class has methods whose behavior varies significantly from one class constant to another, you should use a separate private class or anonymous inner class for each constant. This allows each constant to have its own implementation of each such method and automatically invokes the correct implementation. The alternative is to structure each such method as a multiway branch that behaves differently depending on the constant on which it's invoked. This alternative is ugly, error prone, and likely to provide performance that is inferior to that of the virtual machine's automatic method dispatching.

The two techniques described in the previous paragraphs are illustrated in the typesafe enum class that follows. The class, `Operation`, represents an operation performed by a basic four-function calculator. Outside of the package in which the class is defined, all you can do with an `Operation` constant is to invoke the `Object` methods (`toString`, `hashCode`, `equals`, and so forth). Inside the package, however, you can perform the arithmetic operation represented by the constant. Presumably, the package would export some higher-level calculator object that exported one or more methods that took an `Operation` constant as a parameter. Note that `Operation` itself is an abstract class, containing a single package-private abstract method, `eval`, that performs the appropriate arithmetic operation. An anonymous inner class is defined for each constant so that each constant can define its own version of the eval method:

```
// Typesafe enum with behaviors attached to constants
public abstract class Operation {
    private final String name;

    Operation(String name)   { this.name = name; }

    public String toString() { return this.name; }

    // Perform arithmetic operation represented by this constant
    abstract double eval(double x, double y);

    public static final Operation PLUS = new Operation("+") {
        double eval(double x, double y) { return x + y; }
    };
    public static final Operation MINUS = new Operation("-") {
        double eval(double x, double y) { return x - y; }
    };
```

```
    public static final Operation TIMES = new Operation("*") {
        double eval(double x, double y) { return x * y; }
    };
    public static final Operation DIVIDED_BY =
        new Operation("/") {
            double eval(double x, double y) { return x / y; }
    };
}
```

Typesafe enums are, generally speaking, comparable in performance to `int` enumeration constants. Two distinct instances of a typesafe enum class can never represent the same value, so reference identity comparisons, which are fast, are used to check for logical equality. Clients of a typesafe enum class can use the == operator instead of the `equals` method; the results are guaranteed to be identical, and the == operator may be even faster.

If a typesafe enum class 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 static member class of that top-level class (Item 18). For example, the `java.math.BigDecimal` class contains a collection of `int` enumeration constants representing *rounding modes* for decimal fractions. These rounding modes provide a useful abstraction that is not fundamentally tied to the `BigDecimal` class; they would been better implemented as a freestanding `java.math.RoundingMode` class. This would have encouraged any programmer who needed rounding modes to reuse those rounding modes, leading to increased consistency across APIs.

The basic typesafe enum pattern, as exemplified by both `Suit` implementations shown earlier, is *fixed*: It is impossible for users to add new elements to the enumerated type, as its class has no user-accessible constructors. This makes the class effectively final, whether or not it is declared with the `final` access modifier. This is normally what you want, but occasionally you may want to make a typesafe enum class *extensible*. This might be the case, for example, if you used a typesafe enum to represent image encoding formats and you wanted third parties to be able to add support for new formats.

To make a typesafe enum extensible, merely provide a protected constructor. Others can then extend the class and add new constants to their subclasses. You needn't worry about enumeration constant conflicts as you would if you were using the `int` enum pattern. The extensible variant of the typesafe enum pattern takes advantage of the package namespace to create a "magically administered" namespace for the extensible enumeration. Multiple organizations can extend the enumeration without knowledge of one another, and their extensions will never conflict.

Merely adding an element to an extensible enumerated type does not ensure that the new element is fully supported: Methods that take an element of the enumerated type must contend with the possibility of being passed an element unknown to the programmer. Multiway branches on fixed enumerated types are questionable; on extensible enumerated types they're lethal, as they won't magically grow a branch each time a programmer extends the type.

One way to cope with this problem is to outfit the typesafe enum class with all of the methods necessary to describe the behavior of a constant of the class. Methods that are not useful to clients of the class should be protected to hide them from clients while allowing subclasses to override them. If such a method has no reasonable default implementation, it should be abstract as well as protected.

It is a good idea for extensible typesafe enum classes to override the `equals` and `hashCode` methods with final methods that invoke the `Object` methods. This ensures that no subclass accidentally overrides these methods, maintaining the guarantee that all equal objects of the enumerated type are also identical (`a.equals(b)` if and only if `a==b`):

```
//Override-prevention methods
public final boolean equals(Object that) {
    return super.equals(that);
}

public final int hashCode() {
    return super.hashCode();
}
```

Note that the extensible variant is not compatible with the comparable variant; if you tried to combine them, the ordering among the elements of the subclasses would be a function of the order in which the subclasses were initialized, which could vary from program to program and run to run.

The extensible variant of the typesafe enum pattern is compatible with the serializable variant, but combining these variants demands some care. Each subclass must assign its own ordinals and provide its own `readResolve` method. In essence, each class is responsible for serializing and deserializing its own

instances. To make this concrete, here is a version of the `Operation` class that has been modified to be both extensible and serializable:

```
// Serializable, extensible typesafe enum
public abstract class Operation implements Serializable {
    private final transient String name;
    protected Operation(String name) { this.name = name; }

    public static Operation PLUS = new Operation("+") {
        protected double eval(double x, double y) { return x+y; }
    };
    public static Operation MINUS = new Operation("-") {
        protected double eval(double x, double y) { return x-y; }
    };
    public static Operation TIMES = new Operation("*") {
        protected double eval(double x, double y) { return x*y; }
    };
    public static Operation DIVIDE = new Operation("/") {
        protected double eval(double x, double y) { return x/y; }
    };

    // Perform arithmetic operation represented by this constant
    protected abstract double eval(double x, double y);

    public String toString() { return this.name; }
    // Prevent subclasses from overriding Object.equals
    public final boolean equals(Object that) {
        return super.equals(that);
    }
    public final int hashCode() {
        return super.hashCode();
    }

    // The 4 declarations below are necessary for serialization
    private static int nextOrdinal = 0;
    private final int ordinal = nextOrdinal++;
    private static final Operation[] VALUES =
        { PLUS, MINUS, TIMES, DIVIDE };
    Object readResolve() throws ObjectStreamException {
        return VALUES[ordinal];  // Canonicalize
    }
}
```

Here is a subclass of `Operation` that adds logarithm and exponential operations. This subclass could exist outside of the package containing the revised

Operation class. It could be public, and it could itself be extensible. Multiple independently written subclasses can coexist peacefully:

```
// Subclass of extensible, serializable typesafe enum
abstract class ExtendedOperation extends Operation {
    ExtendedOperation(String name) { super(name); }

    public static Operation LOG = new ExtendedOperation("log") {
        protected double eval(double x, double y) {
            return Math.log(y) / Math.log(x);
        }
    };
    public static Operation EXP = new ExtendedOperation("exp") {
        protected double eval(double x, double y) {
            return Math.pow(x, y);
        }
    };

    // The 4 declarations below are necessary for serialization
    private static int nextOrdinal = 0;
    private final int ordinal = nextOrdinal++;
    private static final Operation[] VALUES = { LOG, EXP };
    Object readResolve() throws ObjectStreamException {
        return VALUES[ordinal];  // Canonicalize
    }
}
```

Note that the `readResolve` methods in the classes just shown are package-private rather than private. This is necessary because the instances of `Operation` and `ExtendedOperation` are, in fact, instances of anonymous subclasses, so private `readResolve` methods would have no effect (Item 57).

The typesafe enum pattern has few disadvantages when compared to the `int` pattern. Perhaps the only serious disadvantage is that it is more awkward to aggregate typesafe enum constants into sets. With `int`-based enums, this is traditionally done by choosing enumeration constant values, each of which is a distinct positive power of two, and representing a set as the bitwise OR of the relevant constants:

```
// Bit-flag variant of int enum pattern
public static final int SUIT_CLUBS    = 1;
public static final int SUIT_DIAMONDS = 2;
public static final int SUIT_HEARTS   = 4;
public static final int SUIT_SPADES   = 8;

public static final int SUIT_BLACK = SUIT_CLUBS | SUIT_SPADES;
```

Representing sets of enumerated type constants in this fashion is concise and extremely fast. For sets of typesafe enum constants, you can use a general purpose set implementation from the Collections Framework, but this is neither as concise nor as fast:

```
Set blackSuits = new HashSet();
blackSuits.add(Suit.CLUBS);
blackSuits.add(Suit.SPADES);
```

While sets of typesafe enum constants probably cannot be made as concise or as fast as sets of `int` enum constants, it is possible to reduce the disparity by providing a special-purpose `Set` implementation that accepts only elements of one type and represents the set internally as a bit vector. Such a set is best implemented in the same package as its element type to allow access, via a package-private field or method, to a bit value internally associated with each typesafe enum constant. It makes sense to provide public constructors that take short sequences of elements as parameters so that idioms like this are possible:

```
hand.discard(new SuitSet(Suit.CLUBS, Suit.SPADES));
```

A minor disadvantage of typesafe enums, when compared with `int` enums, is that typesafe enums can't be used in `switch` statements because they aren't integral constants. Instead, you use an `if` statement, like this:

```
if (suit == Suit.CLUBS) {
    ...
} else if (suit == Suit.DIAMONDS) {
    ...
} else if (suit == Suit.HEARTS) {
    ...
} else if (suit == Suit.SPADES) {
    ...
} else {
    throw new NullPointerException("Null Suit"); // suit == null
}
```

The `if` statement may not perform quite as well as the `switch` statement, but the difference is unlikely to be very significant. Furthermore, the need for multiway branches on typesafe enum constants should be rare because they're amenable to automatic method dispatching by the JVM, as in the `Operator` example.

Another minor performance disadvantage of typesafe enums is that there is a space and time cost to load enum type classes and construct the constant objects. Except on resource-constrained devices like cell phones and toasters, this problem in unlikely to be noticeable in practice.

In summary, the advantages of typesafe enums over `int` enums are great, and none of the disadvantages seem compelling unless an enumerated type is to be used primarily as a set element or in a severely resource constrained environment. Thus **the typesafe enum pattern should be what comes to mind when circumstances call for an enumerated type**. APIs that use typesafe enums are far more programmer friendly than those that use `int` enums. The only reason that typesafe enums are not used more heavily in the Java platform APIs is that the typesafe enum pattern was unknown when many of those APIs were written. Finally, it's worth reiterating that the need for enumerated types of any sort should be relatively rare, as a major use of these types has been made obsolete by subclassing (Item 20).

Item 22: Replace function pointers with classes and interfaces

C supports *function pointers*, which allow a program to store and transmit the ability to invoke a particular function. Function pointers are typically used to allow the caller of a function to specialize its behavior by passing in a pointer to a second function, sometimes referred to as a *callback*. For example, the `qsort` function in C's standard library takes a pointer to a *comparator* function, which it uses to compare the elements to be sorted. The comparator function takes two parameters, each of which is a pointer to an element. It returns a negative integer if the element pointed to by the first parameter is less than the one pointed to by the second, zero if the two elements are equal, and a positive integer if the element pointed to by the first parameter is greater than the one pointed to by the second. Different sort orders can be obtained by passing in different comparator functions. This is an example of the *Strategy* pattern [Gamma98, p. 315]; the comparator function represents a strategy for sorting elements.

Function pointers were omitted from the Java programming language because object references can be used to provide the same functionality. Invoking a method on an object typically performs some operation on *that object*. However, it is possible to define an object whose methods perform operations on *other objects*, passed explicitly to the methods. An instance of a class that exports exactly one such method is effectively a pointer to that method. Such instances are known as *function objects*. For example, consider the following class:

```
class StringLengthComparator {
    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
}
```

This class exports a single method that takes two strings and returns a negative integer if the first string is shorter than the second, zero if the two strings are of equal length, and a positive integer if the first string is longer. This method is a comparator that orders strings based on their length instead of the more typical lexicographic ordering. A reference to a `StringLengthComparator` object serves as a "function pointer" to this comparator, allowing it to be invoked on arbitrary pairs of strings. In other words, a `StringLengthComparator` instance is a *concrete strategy* for string comparison.

As is typical for concrete strategy classes, the `StringLengthComparator` class is *stateless*: It has no fields, hence all instances of the class are functionally

equivalent to one another. Thus it could just as well be a singleton to save on unnecessary object creation costs (Item 4, Item 2):

```
class StringLengthComparator {
    private StringLengthComparator() { }

    public static final StringLengthComparator
        INSTANCE = new StringLengthComparator();

    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
}
```

To pass a `StringLengthComparator` instance to a method, we need an appropriate type for the parameter. It would do no good to use `StringLengthComparator` because clients would be unable to pass any other comparison strategy. Instead, we need to define a `Comparator` interface and modify `StringLengthComparator` to implement this interface. In other words, we need to define a *strategy interface* to go with the concrete strategy class. Here it is:

```
// Strategy interface
public interface Comparator {
    public int compare(Object o1, Object o2);
}
```

This definition of the `Comparator` interface happens to come from the `java.util` package, but there's nothing magic about it; you could just as well have defined it yourself. So that it is applicable to comparators for objects other than strings, its `compare` method takes parameters of type `Object` rather than `String`. Therefore, the `StringLengthComparator` class shown earlier must be modified slightly to implement `Comparator`: The `Object` parameters must be cast to `String` prior to invoking the length method.

Concrete strategy classes are often declared using anonymous classes (Item 18). The following statement sorts an array of strings according to length:

```
Arrays.sort(stringArray, new Comparator() {
    public int compare(Object o1, Object o2) {
        String s1 = (String)o1;
        String s2 = (String)o2;
        return s1.length() - s2.length();
    }
});
```

Because the strategy interface serves as a type for all of its concrete strategy instances, a concrete strategy class needn't be made public to export a concrete strategy. Instead, a "host class" can export a public static field (or static factory method) whose type is the strategy interface, and the concrete strategy class can be a private nested class of the host. In the example that follows, a static member class is used in preference to an anonymous class to allow the concrete strategy class to implement a second interface, `Serializable`:

```
// Exporting a concrete strategy
class Host {
    ...  // Bulk of class omitted

    private static class StrLenCmp
            implements Comparator, Serializable {
        public int compare(Object o1, Object o2) {
            String s1 = (String)o1;
            String s2 = (String)o2;
            return s1.length() - s2.length();
        }
    }

    // Returned comparator is serializable
    public static final Comparator
        STRING_LENGTH_COMPARATOR = new StrLenCmp();
}
```

The `String` class uses this pattern to export a case-independent string comparator via its `CASE_INSENSITIVE_ORDER` field.

To summarize, the primary use of C's function pointers is to implement the Strategy pattern. To implement this pattern in the Java programming language, declare an interface to represent the strategy and a class that implements this interface for each concrete strategy. When a concrete strategy is used only once, its class is typically declared and instantiated using an anonymous class. When a concrete strategy is exported for repeated use, its class is generally a private static member class, and it is exported via a public static final field whose type is the strategy interface.