The Java Language Environment

2.2 Features Removed from C and C++

The earlier part of this chapter concentrated on the principal features of Java. This section discusses features removed from C and C++ in the evolution of Java.

The first step was to *eliminate redundancy* from C and C++. In many ways, the C language evolved into a collection of overlapping features, providing too many ways to say the same thing, while in many cases not providing needed features. C++, in an attempt to add "classes in C", merely added more redundancy while retaining many of the inherent problems of C.

2.2.1 No More Typedefs, Defines, or Preprocessor

Source code written in Java is *simple*. There is no *preprocessor*, no `#define` and related capabilities, no `typedef`, and absent those features, no longer any need for *header files*. Instead of header files, Java language source files provide the declarations of other classes and their methods.

A major problem with C and C++ is the amount of context you need to understand another programmer's code: you have to read all related header files, all related `#define`s, and all related `typedef`s before you can even begin to analyze a program. In essence, programming with `#defines` and `typedef`s results in every programmer inventing a new programming language that's incomprehensible to anybody other than its creator, thus defeating the goals of good programming practices.

In Java, you obtain the effects of `#define` by using constants. You obtain the effects of `typedef` by declaring classes--after all, a class effectively declares a new type. You don't need header files because the Java compiler compiles class definitions into a binary form that retains all the type information through to link time.

By removing all this baggage, Java becomes remarkably *context-free*. Programmers can read and understand code and, more importantly, modify and reuse code much faster and easier.

2.2.2 No More Structures or Unions

Java has no structures or unions as complex data types. You don't need structures and unions when you have classes; you can achieve the same effect simply by declaring a class with the appropriate instance variables.

The code fragment below declares a class called `Point`.

```
class Point extends Object {
    double  x;
    double  y;
            //  methods to access the instance variables
}
```

The following code fragment declares a class called `Rectangle` that uses objects of the `Point` class as instance variables.

```
class Rectangle extends Object {
    Point  lowerLeft;
    Point  upperRight;
            //  methods to access the instance variables
}
```

In C you'd define these classes as structures. In Java, you simply declare classes. You can make the instance variables as private or as public as you wish, depending on how much you wish to hide the details of the implementation from other objects.

2.2.3 No Enums

Java has no *enum* types. You can obtain something similar to `enum` by declaring a class whose only *raison d'etre* is to hold constants. You could use this feature something like this:

```
class Direction extends Object {
    public static final int North = 1;
    public static final int South = 2;
    public static final int East  = 3;
    public static final int West  = 4;
```

```
}
```

You can now refer to, say, the `South` constant using the notation `Direction.South`.

Using classes to contain constants in this way provides a major advantage over C's `enum` types. In C (and C++), names defined in `enum`s must be unique: if you have an `enum` called `HotColors` containing names `Red` and `Yellow`, you can't use those names in any other `enum`. You couldn't, for instance, define another `Enum` called `TrafficLightColors` also containing `Red` and `Yellow`.

Using the class-to-contain-constants technique in Java, you can use the same names in different classes, because those names are qualified by the name of the containing class. From our example just above, you might wish to create another class called `CompassRose`:

```
class CompassRose extends Object {
    public static final int North     = 1;
    public static final int NorthEast = 2;
    public static final int East      = 3;
    public static final int SouthEast = 4;
    public static final int South     = 5;
    public static final int SouthWest = 6;
    public static final int West      = 7;
    public static final int NorthWest = 8;
}
```

There is no ambiguity because the name of the containing class acts as a qualifier for the constants. In the second example, you would use the notation `CompassRose.NorthWest` to access the corresponding value. Java effectively provides you the concept of qualified `enum`s, all within the existing class mechanisms.

2.2.4 No More Functions

Java has no *functions*. Object-oriented programming supersedes functional and procedural styles. Mixing the two styles just leads to confusion and dilutes the purity of an object-oriented language. Anything you can do with a function you can do just as well by defining a class and creating methods for that class. Consider the `Point` class from above. We've added public methods to set and access the instance variables:

```
class Point extends Object {
    double  x;
    double  y;


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

If the `x` and `y` instance variables are private to this class, the only means to access them is via the public methods of the class. Here's how you'd use objects of the `Point` class from within, say, an object of the `Rectangle` class:

```
class Rectangle extends Object {
    Point  lowerLeft;
    Point  upperRight;


    public void setEmptyRect() {
        lowerLeft.setX(0.0);
        lowerLeft.setY(0.0);
        upperRight.setX(0.0);
        upperRight.setY(0.0);
    }
}
```

It's not to say that functions and procedures are inherently wrong. But given classes and methods, we're now down to only one way to express a given task. By eliminating functions, your job as a programmer is immensely simplified: you work *only* with classes and their methods.

2.2.5 No More Multiple Inheritance

Multiple inheritance--and all the problems it generates--was discarded from Java. The desirable features of multiple inheritance are provided by *interfaces*--conceptually similar to Objective C protocols.

An interface is not a definition of a class. Rather, it's a definition of a set of methods that one or more classes will implement. An important issue of interfaces is that they declare only methods and constants. Variables may not be defined in interfaces.

2.2.6 No More Goto Statements

Java has no `goto` statement[1]. Studies illustrated that `goto` is (mis)used more often than not simply "because it's there". Eliminating `goto` led to a simplification of the language--there are no rules about the effects of a `goto` into the middle of a `for` statement, for example. Studies on approximately 100,000 lines of C code determined that roughly 90 percent of the `goto` statements were used purely to obtain the effect of breaking out of nested loops. As mentioned above, multi-level `break` and `continue` remove most of the need for `goto` statements.

2.2.7 No More Operator Overloading

There are no means provided by which programmers can overload the standard arithmetic operators. Once again, the effects of operator overloading can be just as easily achieved by declaring a class, appropriate instance variables, and appropriate methods to manipulate those variables. Eliminating operator overloading leads to great simplification of code.

2.2.8 No More Automatic Coercions

Java prohibits C and C++ style *automatic coercions*. If you wish to coerce a data element of one type to a data type that would result in loss of precision, you must do so explicitly by using a cast. Consider this code fragment:

```
int  myInt;
double  myFloat = 3.14159;

myInt = myFloat;
```

The assignment of `myFloat` to `myInt` would result in a compiler error indicating a possible loss of precision and that you must use an explicit cast. Thus, you should re-write the code fragments as:

```
int  myInt;
double  myFloat = 3.14159;

myInt = (int)myFloat;
```

2.2.9 No More Pointers

Most studies agree that *pointers* are one of the primary features that enable programmers to inject bugs into their code. Given that structures are gone, and arrays and strings are objects, the need for pointers to these constructs goes away. Thus, Java has no pointer data types. Any task that would require arrays, structures, and pointers in C can be more easily and reliably performed by declaring objects and arrays of objects. Instead of complex pointer manipulation on array pointers, you access arrays by their arithmetic indices. The Java run-time system checks all array indexing to ensure indices are within the bounds of the array.
You no longer have dangling pointers and trashing of memory because of incorrect pointers, because there are no pointers in Java.

CONTENTS | PREV | NEXT