A "D" in Programming, Part 1

by Chuck Allison

ISTOCKPHOTO

They say there are three things you should never discuss if you want to get along with other people: politics, sports, and religion. Since I want to stay on your good side, I'll steer clear of these topics, but I would like to discuss something else that often elicits religious-like fervor from developers—favorite programming languages.

Language Wars

What's the best programming language? Be honest—how many times have you defended your favorite language du jour? And how many languages have been your "favorite" at one time or another?

To seed the discussion, let me relate my first and only exposure to COBOL, one of my "unfavorites." In 1975, I was standing in line, card deck in hand, waiting to submit my job to the IBM 360 expediters at Brigham Young University. I had about forty cards containing a Fortran program that numerically solved a system of ordinary differential equations. Complex stuff. Behind me stood another student with a deck of cards more than a foot thick—800 to 1,000 cards, I'd guess. Impressed and intimidated, I asked the student what his program did. "It writes a report." Hmmm. Since I had written reports in Fortran in considerably fewer statements, I decided right then and there I would never use COBOL. No offense to the many COBOL aficionados out there—I just knew it wouldn't work for me. My mathematical background has ingrained within me a constant yearning for economy of expression.

One language is not necessarily better than another, of course. For example, I prefer Python to Perl, but both are fine scripting languages as well as general-purpose, multi-paradigm programming languages. I see Python as cleaner and more orthogonal in its design—your mileage may vary. Nonetheless, you may appreciate the following "Jedi wisdom":

> **YODA:** Code! Yes. A programmer's strength flows from code maintainability. But beware of Perl. Terse syntax … more than one way to do it … default variables. The dark side of code maintainability are they. Easily they flow, quick to join you when code you write. If once you start down the dark path, forever will it dominate your destiny, consume you it will.
>
> **LUKE:** Is Perl better than Python?
>
> **YODA:** No … no … no. Quicker, easier, more seductive.
>
> **LUKE:** But how will I know why Python is better than Perl?
>
> **YODA:** You will know. When your code you try to read six months from now. [1]

Not too many years have passed since the Java "revolution," fueled not only by Sun's leveraging the sudden popularity of the Internet but also in part as a "reaction" to C++. I well remember public discussions at conferences and online debunking C++ as unsafe ("pointers are evil and Java doesn't have pointers"—a deception) and too difficult to learn (Java is no cakewalk either). These comments were relentlessly proffered as context for evangelizing the so-called superiority of this new programming language. I respect Java and its inventors, have used the language professionally and taught it in academia, but I still wince when I recall the bad form that characterized Java's rise to popularity. There is a certain irony in the fact that the HotSpot JVM is written in C++.

That said, there is always room for improvement. Java does have a simpler object model and a more flexible execution model than C++. It has brought portable concurrency and GUI tools to Joe Programmer. The relatively new Scala language is in turn an improvement on Java, combining high-level, functional programming constructs with static type safety. Scala translates to Java byte code providing access to the massive Java library. The Groovy language is a similar alternative but with dynamic typing.

Alas, it's been twelve years since Java made its debut, and its popularity seems to have peaked as it is finding its niche mainly in enterprise applications. Just today I found the following in an *InfoWorld* article entitled, "Java is Becoming the New COBOL":

> "Java, the oldest new programming language around, is falling out of favor with developers. When it comes to developing the increasingly common rich Internet applications, Java is losing ground to Ruby on Rails, PHP, AJAX and other cool new languages … Now that Java is no longer the unchallenged champ for Internet-delivered apps, it makes sense for companies to find programmers who are skilled in the new languages. If you're a Java developer, now's the time to invest in new skills." [2]

New Favorite

In recent Code Craft articles, I praised the relatively new D programming language. D combines the speed and power of C++ with desirable features of many other programming languages, such as Java, C#, Eiffel, and functional programming languages. D is high-level, readable yet concise, multi-paradigm, and efficient. In this article and the next I'll conclude (for now) my D evangelizing by exploring a few of its more noteworthy features.

Foremost for me is the fact that D is strongly typed, compiles to native code, and yet is garbage collected. Currently this is a singular achievement in the programming world. But there is much more about D to recommend.

Code Consistency with Scope Guards

Unless carefully planned for, runtime errors can easily place a program in an inconsistent state. Resource management is a typical example. Suppose you have a situation similar to the function in listing 1.

A Java-like solution uses a `finally` block similar to the D program in listing 2.

A C++-style approach encapsulates resource management into an object's constructor and destructor. This is the well-known Resource Acquisition Is Initialization (RAII) idiom, shown in listing 3.

This approach makes `f`'s code concise, but it also requires an auxiliary class.

D supports RAII, but it also lets you explicitly supply specific cleanup code for when execution exits a scope, as shown in listing 4.

The `scope` statement, called a *scope guard*, activates a code block that may or may not run when a scope is exited. Figure 1 lists the three scope-guard options.

`scope(exit)`	the code always runs (like `finally`)
`scope(failure)`	the code runs only if an exception occurs
`scope(success)`	the code runs only if no exception occurs

Figure 1

The utility of the `scope` statement becomes more obvious with multi-step resource acquisition requiring rollback semantics in the case of failure. Consider the three-part transaction in listing 5.

In this case, we want all three operations to succeed or fail together. RAII alone will not solve this, and the `try-finally` approach is not pretty—witness listing 6.

Things only get worse with more pieces in the transaction. Not so with D. Listing 7 shows how easily you can back out of complicated transactions with D's scope statement.

When execution leaves a scope, all scope-guard blocks that have executed are visited in last-in-first-out order, so transactions roll back gracefully, and thirteen lines of code become six. If `risky_op2()` is the first to fail, only `undo_risky_op1()` executes as the execution propagates out of `g`. Likewise, if only `risky_op3()` fails, then `undo_risky_op2()` executes, followed by `undo_risky_op1()`.

```
void f() {
    acquire();
    risky_op(); // Might fail
    release();
    writeln("f succeeded");
}
```

Listing 1

```
void f() {
    acquire();
    try {
        risky_op();
        writeln("f succeeded");
    }
    finally {
       release();
    }
}
```

Listing 2

```
class Resource {
public:
    Resource() {
       cout << "resource acquired\n";
    }

    ~Resource() {
       cout << "resource released\n";
    }
};

void f() {
    Resource r;
    risky_op();
    cout << "f succeeded\n";
}
```

Listing 3

```
void f() {
    acquire();
    scope(exit) release();
    risky_op();
    writeln("f succeeded");
}
```

Listing 4

```
void g() {
   risky_op1();
   risky_op2();
   risky_op3();
   writeln("g succeeded");
}
```

Listing 5

In Part 2, I'll show how programming with nested functions and delegates can be more concise and powerful than classes and objects. **{end}**

REFERENCES:

1] www.netfunny.com/rhf/jokes/99/Nov/perl.html

2] www.infoworld.com/article/07/12/28/52FE-underreported-java_1.html


```
void g() {
    risky_op1();
    try {
        risky_op2();
    }
    catch (Exception x) {
        undo_risky_op1();      // Back-out op1
        throw x;               // Rethrow exception
    }
    try {
        risky_op3();
        writeln("g succeeded");
    }
    catch (Exception x) {
        // Back-out op1 and op2 in reverse order
        undo_risky_op2();
        undo_risky_op1();
        throw x;
    }
}
```

Listing 6

```
void g() {
    risky_op1();
    scope(failure) undo_risky_op1();
    risky_op2();
    scope(failure) undo_risky_op2();
    risky_op3();
    writeln("g succeeded");
}
```

Listing 7

What have been your favorite languages over the years? What is your opinion of D's scope statement?

Follow the link on the StickyMinds.com homepage to join the conversation.