The Blue programming language
Simplicity and minimalism

M. Tim Jones
August 11, 2009

Languages are the means by which we express our desires to computers systems, and, as far as I'm concerned, there's no such thing as too many. One unique language, called Blue, is an open source object-oriented language that is multipurpose and intuitive to use. This tip provides the foundation for Blue and shows you how to build simple networking applications.

As a software engineer, I've always been fascinated by programming languages. Whether they're your typical imperative language (such as C), a functional language (such as Scheme), or an object-oriented language (such as Ruby), there's something fascinating about the grammar, design, and implementation of languages. Like natural languages, programming languages are the means by which we communicate with machines to specify our intent. But not all languages are the same, and each provides interesting and distinct ways to solve problems.

One such language that provides a unique aspect to problem-solving is Blue. Blue is a dynamically typed object-oriented language that is also multi-paradigm — meaning that it supports paradigms from multiple language styles (for example, mixing object-oriented with functional language aspects). This is useful from a problem-solving perspective because it allows you to mix styles, solving individual problems in the most effective way.

Multi-paradigm languages

While multi-paradigm languages are not necessarily new, they certainly add to the expressiveness of modern programming languages. Some of the common multi-paradigm languages include C++, Ruby, Ada, Python, the Java™ programming language, and many others. Some use stylings from two or three paradigms, while others use as many as eight (such as the Oz language from the Mozart programming system).

Prior to delving into a more technical introduction to Blue, it's important to note that this tip focuses on the Blue programming language developed by Erik Lechak, not the Blue programming language developed at Monash University.

Motivation for Blue

Blue is a dynamic programming language with unique features. It was designed as a cross-platform multi-paradigm language with a simple and consistent syntax. Blue is interpreted, but
can be compiled to bytecode for greater performance. But what is most striking about Blue is its focus on minimalism. It presents a rich, expressive language with minimal features. For example, in many languages, there are a number of ways in which iteration is implemented. In Blue, ignoring recursion or map (functional iterator), there’s one way to implement loop constructs.

Language features
While a complete introduction to Blue isn’t possible in this tip, looking at some of the more notable features will help you better understand the language. This section explores types, conditions, iterations, error trapping, and classes, followed by a couple of slightly larger Blue applications to pull the elements together.

Types
Blue includes a number of built-in types, including numbers (which covers integer and floating point types), strings, arrays, functions, and others. Recall that Blue is dynamically typed, which means you simply associate a variable with a value (variable definition is not required). The example in Listing 1 illustrates variable creation of a variety of types (int, float, string, array, and function).

Listing 1. Variable creation of a variety of types

```blue
x = 9;
y = 3.14159;
s = "this is a string";
a = [1, 2, 3, 5, 8];
f = func{ arg name; sys.print( "Hello ", name, "\n" ); };
```

The last line emits the attributes for a given string type. These attributes are the native type attributes (or members) of that type. When the `attribs` function is executed on the type of the variable — which is retrieved using the `typeof` operator (`&`) — the result is:

```
[replace, repeat, num, ltrim, length, import, endsWith, compile, find, eval, load, print, substr, rtrim, save, startsWith, split, trim]
```

These methods should look familiar as typical string object methods. You can also use the `attribs` method on the object itself (rather than the type) which provides the attributes of the object itself.

Conditions
In Blue, a condition is an expression instead of a statement. This means you can use a condition in place of a value, which can result in simpler and cleaner code. In Blue, a condition takes the form:

```
condition ? true-expression ;
condition ? true-expression : false-expression ;
```

So, you start with an expression and provide one or two options (true or false paths). Two examples are provided in Listing 2. In these simple cases, expressions are provided, but they could also be blocks of code (using curly braces) to specify multiple statements.
### Listing 2. Examples of conditions

```plaintext
foo = 2;
(foo == 2) ? sys.print("foo is 2\n");
(foo == 1) ? sys.print("foo is 1\n") : sys.print("foo is not 1\n");
```

### Iteration

Blue provides a single mechanism for looping, which is contained within a code block. A code block is simply a block of code that returns a value, and if no value is returned then a NULL object is returned. A code block is defined as shown below.

#### Listing 3. How a code block is defined

```plaintext
z = {
    x = 1;
    y = x+3;
    return y;
};
sys.print( z );
```

An iteration is built from this pattern as shown in Listing 4 (accumulating the numbers from 10 to 1):

#### Listing 4. Example iteration

```plaintext
sum = 0;
value = 10;
ret = loop {
    sum = sum + value;
    value = value - 1;
    (value == 0) ? return sum;
};
sys.print( ret );
```

In this example, the code block is iterated until the loop is broken through the condition. But even with the simplicity of this loop construct, you can build any complex loop.

### Trapping errors

Blue implements an exception system that is similar to the Java programming language in some ways, but with some simplifications. In Blue, you "raise" an exception (compared to Java's "throwing" of an exception). You can also "catch" exceptions to manage errors in Blue with the error-trapping operator (\(|\)). This section provides a few examples to show how this works.

The most common use of exception handling is breaking out of a function to raise an exception to a caller (which may be the caller or a caller higher up the stack). In the contrived example shown in Listing 5, I validate the passed argument, and if it is invalid, I raise an exception. One difference in Blue is that you can raise any object as an exception. At the caller, I use the error-
trapping operator to catch the exception. I emit an error message and replace the return value with 0, instead (remember: every inline code block returns an object).

**Listing 5. Example of exception handling**

```
z = func {
    arg my_arg;
    (my_arg == 0) ? raise -1;
    return 10 / my_arg;
};
x = z(0) | { sys.print("error\n"); return 0; }
```

Another interesting and simpler use case for this method is in providing default values for arguments. You can use the line of code shown in Listing 6 to specify a default value for an argument that is absent. In this case, if the user calls function `z` without an argument, 1 is used as a default value.

**Listing 6. Specifying a default value for an absent argument**

```
z = func {
    arg my_arg | 1;
    ...
};
```

Finally, you can implement the traditional use case for exception handling over a block of code as shown in Listing 7. This is a typical use case that covers the entire block for managing exceptions.

**Listing 7. Exception handling over a block of code**

```
{ first_func();
  second_func();
  ...
} | {
    # Manage errors for the block
}
```

**Classes**

Blue also takes a clearly minimalist approach to object-oriented programming. In Blue, you can create new classes, inherit from classes, or extend classes. To create a new class, I begin with the `sys.class()` function. This returns a new empty class to which I can add attributes. Listing 8 explores the creation of a character class that could be used in a simple role-playing game. I can now fill the new empty class with attributes such as string and number types (representing the elements of the character), as well as functions common to this class. Note that I provide three functions: a constructor, a destructor, and a status function that emits information about the character represented by the object instance.
Listing 8. Character class

```
# Character Class
# Character = sys.class();

Character.name = "";
Character.weapon = "";
Character.hp = 0;

# Constructor
Character._ = func {
    arg name | { name = "anonymous"; };
    arg weapon | { weapon = "stick"; };
    arg hp | { hp = 10; };

    this.name = name;
    this.weapon = weapon;
    this.hp = hp;
};

# Destructor
Character.__ = func {
    sys.print( this.name, " is dead.\n" );
};

# Class method
Character.status = func {
    sys.print( this.name, " (HP ", this.hp, ") wields a ", this.weapon, ".\n" );
};
```

With the character class complete, I can now create character objects, as shown in Listing 9. In the first case, I create an object, `char1`, and provide all the arguments. In the second case, I don't provide any arguments, so it uses the defaults. Finally, I call the status functions for each of the objects.

Listing 9. Character objects

```
char1 = Character( "Ralph", "whip", 5 );
char2 = Character();

char1.status();
char2.status();
```

But Blue is highly dynamic, so you can amend objects after you create them. In the example shown in Listing 10, I add a `greet` function to the `char1` object. This function is specific to `char1` and appears in no other instances of the `Character` class.

Listing 10. Adding a `greet` function to the `char1` object

```
char1.greet = func {
    sys.print( this.name, " says hi.\n" );
};
char1.greet();
del char1.greet();
```

The output for this application, including the three prior listings, is shown below.
Though not shown in these examples, Blue does allow for inheritance. Blue even allows you to extend classes dynamically and contract them (that is, remove class attributes from an extended class).

**Networking example**

This section provides an example of Blue in the form of a networking application. One of the interesting aspects of most object-oriented scripting languages is their ability to prototype networking applications quickly and easily. Blue is no exception. Blue implements modules for useful behaviors such as sockets, threads, streams (of various kinds), and even a simple XML parser.

This final example demonstrates a simple dynamic Web server. *Dynamic* means that the pages are constructed at runtime instead of reading content from the filesystem. The simple Blue Web server is shown below.

**Listing 12. Simple dynamic Web server**

```
# Dynamic Web Server Class
#
WebServerClass = sys.class();
WebServerClass.name = "";
WebServerClass.port = 0;

# Constructor
WebServerClass._ = func {
    arg name | { name = "SimpleWebServer"; };
    arg port | { port = 80; };
    this.name = name;
    this.port = port;
    global Stream = sys.library("streams.dll");
};

# Server
WebServerClass.server = func {
    server = Stream.listen(this.port);
    loop {
        client = server.accept();
        request = client.read().substr("/", "");
        sys.print("Request: ", request, "\n");
    }
};
```

Listing 11. Output for application that includes Listings 8-10

```
$ blue char.bl
Ralph (HP 5) wields a whip.
anonymous (HP 10) wields a stick.
Ralph says hi.
```
{  
    client.write("<html><head><title>");  
    client.write( this.name );  
    client.write( "</title></head><body>" );  
    ( request.find("status") != 0 ) ? {  
        client.write("<h1>Server is up.</h1>" );  
    } ;  
} | {  
    # Exception Handler  
    client.write("<h1>Page not found.</h1>" );  
} ;  
client.write("</body></html>" );  
client = ();}

myWebServer = WebServerClass( "MyWebServer", 80 );  
myWebServer.server();

The example begins by creating a new class called WebServerClass. I add a string name and port class variable and then two functions: a constructor and the server. I use the constructor to initialize the arguments and perform any required setup (such as loading the streams library). The server function then implements the Web server loop. This function creates a listening socket on the defined port and waits for a server to connect through the accept function.

When a client has connected, the HTTP server reads the HTTP request and provides a response through the client socket. The request indicates the file requested, and that file is tested using the find function. If the status file is requested, a simple message is emitted. Otherwise, a trap occurs for find, and the exception handler emits the standard file not found error. When the output is complete, the stream is flushed by writing a NULL object to the client stream and loop awaiting the next client. The end of the listing shows how to create and start a WebServerClass object.

Language status

Blue is under active development, with its latest release (V1.75) having arrived in April 2009. You can use Blue on Linux® and Microsoft® Windows®, with support for OS X coming soon. The Blue Web site provides a forum for language discussion, current and past tarballs for installation, and tutorials on language features (see Resources).

Moving forward

Blue is an interesting language that has a number of compelling concepts. Like many of the object-oriented scripting languages (Ruby, Python, etc.), it is easy to learn and enjoyable to use.
This class of language is ideal for quick prototyping and testing new ideas. Blue is still under development and has the potential to change, but it's well worth your while to check it out.

© Copyright IBM Corporation 2009
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)