Column #86 June 2002 by Jon Williams:

Objects in the Machine

A long time ago, in a galaxy far, far away .... Actually, it was the early 90's and it was in Palo Alto, California – at about the same time the BASIC Stamp was being developed – the guys a Sun Microsystems were working on a little language called Oak that they hoped would change the way we program microcontrollers. The idea was to create a clean, easy, object-oriented language that could be used on a variety of small micros; the concept that one program would work on any micro – cross platform compatibility was the goal.

Due to a naming conflict with another product, Oak was changed to Java. Coincident with the later stages of its development was the public emergence of the Internet. The nature of the Internet and the variety of operating systems connected to it was ideally suited for the cross-platform approach designed into Java. Java's target changed and it became a contributing factor the wild success and proliferation of the Internet and World Wide Web.

Pity, really, that they didn't keep working towards the small micros since Java is a very nice language. It has, in fact, garnered tremendous acceptance from programmers and recently with educational institutions. Most universities and many high schools are switching from teaching C and C++ to teaching Java. Well, it took a while, but Java is finally available in a microcontroller format.

Parallax isn't the first to do it, but just as the case was with the BASIC Stamp, Parallax has made it clean and easy with the Javelin Stamp. So if you're a C/C++ or Java programmer wanting to explore the world of embedded control, now you can do with a product from a company famous for its reliable products and after-sale support – and you can do it with a cool little 24-pin DIP package.

This article is written for those PBASIC programmers who are new to Java, bought a Javelin for all its cool features and power, and have worked your way through the programming tutorials in the manual. If you don't have a Javelin starter kit yet, that's okay too. Just be sure to download the manual (it's free) and read through chapters 2, 3 and 4 – or none of this will make any sense.

Let's say that you do have a Javelin and have played with the tutorial examples. Now you're excited, now you want to create your own classes ... but wait, this isn't quite as straightforward as programming a BASIC Stamp. No, it isn't, but you know what? Taking the time and putting the effort to learning Java properly is worthwhile, especially as you start to fill that 32K code space and take on increasingly sophisticated tasks.

OOP for Regular Guys (and Gals!)

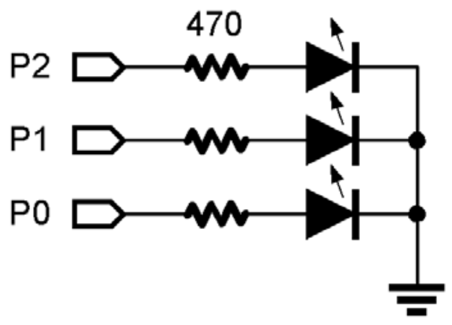
You've probably heard the term Object Oriented Programming (OOP) before, but what does it mean, exactly? It becomes self-evident when we understand the programmer's definition of an object. An object is a [software] mechanism for bundling data (variables) and the means to manipulate that data (code). This bundling is called encapsulation. In procedural (action-oriented) languages like BASIC, data and code are separated. Not so with Java. In Java, objects contain both data and the code (behaviors) necessary to manipulate or use the data. They are fully self-contained. In a Java program, objects pass messages to each other. This is the nature of Object Oriented Programming.

Why should we care about this? Well, for one thing it can make things easier by hiding gritty details (this is especially nice when you're using code written by someone else). The biggest advantage, though, is that it makes reusing code very easy – saving lots of time for programmers. We'll see both these concepts in action with our little demo program.

Let's Just Do It

The easiest way to get your arms around Object Oriented Programming is to just haul-off and do it. So we will. We'll keep it simple so that we can grasp Java concepts and get used to the way the Javelin does things. Since all programming tutorials start off with a "Hello World" program and with microcontrollers we say "Hello" by flashing an LED, that's what we'll do. Yes, yes, I know it sounds amazingly trivial but I promise that the exercise is a good one.

Figure 86.1: Javelin Stamp to LEDs Schematic



Most of us don't think twice about LEDs because we've used them so many times. An herein lies one of the keys to succeeding with an OOP language like Java: thinking about it – a lot – at first will save you hours of code rewriting later. OOP programs need to be designed, not simply written. An LED is a good place to start with OOP, since it's a real-world object that that is easy to model with software.

It becomes pretty interesting when we actually stop to think about what concerns we could have when connecting an LED to a microcontroller. What pin will we use? How will we control it? When we know these things then we can create code (called methods in Java) to turn it off, turn it on, control it indirectly, invert its current state and to report what it's doing at the moment.

Before we look at the LED code, let me remind you that all Java programs are composed of one or more class files. To create an object definition, we'll write a class file with the same name as the object type we're defining. It's important to distinguish the class file (definition) from the actual instance of an object. My friend, Al, told me to think of a object's class file like a cookie-cutter. It isn't the object itself, it's what gives the object its "shape" – so to speak. What you'll find as you spend more time with Java is that one class can be extended to create a more complex version. This is what is meant by inheritance: the new class inherits all the capabilities of the class used to build it.

Time to jump in and take a look at Program Listing 86.1, the LED class. From this definition we'll be able to create LEDs to our heart's content and control them with ease.

One of this first things you notice is an import line. What this is doing is importing (gaining access to) all (*) of the classes in the core package [folder]. The classes in this package have to do with the Javelin's hardware, specifically the CPU class. We'll be using a couple methods from the CPU class to control control the pin that controls our LED.

At the top, we define a couple of constant (final) values to indicate the current state of the LED. The constants have values of true and false, respectively, because and LED is either on (true) or off (false). These constants are public so they will be the same for every LED object we create.

Next come a few private variables (called fields in Java). It's important that these values are private because we need them to be different for every LED object we create. And being private, they're hidden from the outside world. In order to change these variables, we'll create methods to access and manipulate them.

In a tangible class (there are abstract classes that are used as the base for others) like LED, the first method we'll deal with is special; in fact it has the same name as the class. This method is called the constructor. The constructor does the work required when creating a working object. Some classes have multiple contructors (using a technique called method overloading) that allow an object to be create my more than one means. We'll keep it simple and just have one constructor in our LED class.

In our case we're going to set the pin to use, how we control the LED, then make sure it's off. Notice the use of the word "this." While not strictly necessary, it does help clarify the code a bit. The use of "this" indicates the object's fields, not the parameters passed. The final line of the constructor calls the LED.off() method to make sure the LED starts in a known state. This is important since the pin may have been in use before defined to control the LED.

Now we can add methods to the LED class that actually perform the control we discussed earlier: turn it off, turn it on, follow an external value (indirect control), invert its state and report what it's currently doing.

The first method is called, big surprise, off(). This method works by calling the CPU.readPin() method. Remember that CPU.readPin() method is normally used to report the state of an input pin, so it starts by making the specified pin an input. In our case, this will safely extinguish the LED. We save the new state (off) so it's easy to retrieve later and we don't have to rely on examining hardware for the current LED state.

Our next method is used to turn the LED on. This uses the CPU.writePin() method which, of course, is used to make a pin an output and set it high or low. To set a pin high, we pass a value of true; to set it low we pass a value of false. In our case, the pin will be the LED connection we passed to the constructor and the control value will be what we set in the onState field. Once again, we save the new state (on) so we can refer to it later.

There may be times when we want the LED to follow the value of another variable or logical condition so a method that will allow that is convenient. This is the purpose of the putState() method. This method requires a Boolean value (true or false) to be passed passed to it. If the value passed is true, the LED will be lit. If false, it will be extinguished. This is very easy code since we've already written methods to turn the LED on and off. Note that the value passed is not related to how we control the LED – true means on, false means off; no matter how the LED is connected.

An easy way to blink an LED is to invert its state inside a loop with a bit of a delay, so let's create an invert() method. This method sends the inverted (!) ledState value to the putState() method. Simple, huh?

Finally, there will be times when we want to know if an LED is on or off – without remembering and dealing with the connection and control details. This is the purpose of the getState() method. Notice that this method is defined as a Boolean type since it returns a value of true or false. The other methods we've created don't return anything so they are defined as void. Since we've previously saved the state of our LED, it's a simple matter of returning it to the caller.

Now, to those of you who are PBASIC fanatics (like me), this may seem like an amazing amount of work just to manipulate an LED. It only seems that way now. What you'll find after you've worked with the Javelin a bit is that this up-front work makes the down-the-road stuff a breeze, especially when it comes to bigger programs and more complex objects.

LEDs In Action

Okay, time to make this stuff work and show the elegance of Java in action. The circuit is a no-brainer: three LEDs and three resistors – probably stuff you have within an arm's reach of you computer.

The working code is in Program Listing 86.2. As with all Java programs, this is a class file too, but this class doesn't define an object. As I stated earlier, a Java program is made up of one or more classes. Java knows where to start because it always starts with the method called main(). So, we'll have a main() method in this class since it's this code we want to run.

As with most of our Javelin programs, we'll import the "core" classes and, of course, we'll import our new LED class so we can use it.

Now we get to the good stuff – using our LED class. Using standard Java syntax and referencing our new object class, we'll create three LED objects: one green, one yellow and one red. These are simple names for our demo. Take advantage of Java's verbose nature and name your objects and variables so that your programs are easy to read. An HVAC project, for example, might have LED objects called acCooling and acHeating.

At the end of our demo we're going to do some precision timing so we'll define a timer for each – and give them an appropriate name. Timers use a special Virtual Peripheral (VP) that runs in the background. The Javelin has several virtual peripherals and up to six can run at the same time. The nice thing about timers is that only one VP slot is used, no matter how many timers we define. Other VPs include UARTs for background serial communications, a PWM VP that's perfect for servo and motor control, A2D and D2A conversion.

Back to our LEDs; let's start with the basics. And now we see the fruits of the work we went through to design the LED class. How easy is this?: green.on(). Could we make it any more obvious? If so, I don't know how. And this is part of the point. Well-designed code is easy to read and understand without a lot of additional comments. That applies to any programming language, but Java is particularly supportive of this concept.

Since we know how our object code works and the rest of the basic demo is easy, let me just explain that the CPU.delay() method is identical to PBASIC's PAUSE command except that takes a parameter in 100 microsecond (0.1 millisecond) units. So, a delay of 5000 units is a half second.

Now for the timers, because these are incredibly useful. Once we've defined a timer, we start it with the mark() method. We can check to see if a given timer has reached an elapsed time (in milliseconds) with the timeout() method that passes a single parameter. This will return true if the timer value is equal to or greater than the value passed (there is another version of the timeout method that allows even greater resolution).

So here's what's happening at the end of the code. We've created a very tight while-loop that checks on the timers for each LED. When a timer expires (timeout () returns true), we restart that timer and invert the state of the associated LED. In the end, you'll see all three LEDs flashing at completely independent rates. This is a neat demo, right out of Javelin manual. But now extend this idea beyond flashing LEDs. How many of your projects could benefit from launching specific chunks of code at precise timing intervals? A lot, I'll bet.

In the end, the question begs to be asked... Did we really need to create an class file to control an LED? Of course not. That said, which line of code do you think is more obvious?:

```
alarmLED.on();
```

or...

```
CPU.writePin(CPU.pin0, true);
```

Both are valid. Both work. Only one tells us exactly what's going on. Java requires a little bit of discipline and yet it pays tremendous dividends for doing so. Besides, it you don't have time to do it right, when will you find time to do it again?

Where Do We Go From Here?

If you've worked your way through the manual and started to play, you might do what I've been doing to learn the Javelin: I've been porting some of my favorite BASIC Stamp projects and code snippets – especially those things I tended to use a lot as they're perfect candidates for object class files. If you want more reference material, there are over a 1000 books available on Java programming. Almost none of them deal with embedded programming, per se, but they do teach Java well and most of the code will run with little or no modification on your Javelin.

Also keep an eye on the Javelin the web site for new code and application notes. As Parallax (and others) develop new class files, they will be documented and posted. This will, ultimately, save us all a lot of work and allow us to focus on designing solutions instead of churning out grunt code. And make sure you join the Javelin Stamp group (on YahooGroups). That will keep you in touch with other Javelin programmers.

I think you'll agree that the Javelin is very exciting. If we look back and see all the cool things the BASIC Stamp has done, one can only imagine what the Javelin and its programmers will do....

How Fast Did You Say?

The readers of this magazine are sharp. Very sharp. Several caught my horribly glaring error in last month's column.

The correct speed of sound is, of course, about 1130 feet per second at sea level – in dry air at 72 degrees Fahrenheit. That's a far cry faster than the 0.9 feet per second that I quoted. Please accept my apology for the error. I was referencing an old [incorrect] SRF-04 tech sheet. Still, it's my responsibility to verify technical data that I don't generate myself. I'll do that in the future.

To me, the only thing more appealing than an intelligent person is an intelligent person with a sense of humor. One reader suggested that the reason Texans apparently speak more slowly than the rest of the world could be accounted for by my speed of sound error. Funny. Very funny. Truth be told, I'm a native Californian currently living in the great state of Texas – and I wouldn't have it any other way.

Until next month, happy Stamping – in PBASIC or Java!

Program Listing 86.1.

```
package stamp.peripheral.io;

import stamp.core.*;

/**
 * This class encapsulates the basic operations of a standard LED.
 *
 * @author Jon Williams, Parallax Inc.
 * @version 1.0 03 April 2002
 */
public class LED {

  public static final boolean LED_OFF = false;
  public static final boolean LED_ON  = true;

  private int ioPin;
  private boolean onState;
  private boolean ledState;

  /**
   * Creates an LED object.
   *
   * @param ioPin LED control pin
   * @param onState Output state of control pin to light LED
   */
  public LED (int ioPin, boolean onState) {
    this.ioPin = ioPin;
    this.onState = onState;
    off();
  }

  /**
   * Extinguishes the LED
   */
  public void off() {
    CPU.readPin(ioPin);
    ledState = LED_OFF;
  }

  /**
   * Lights the LED
   */
  public void on() {
    CPU.writePin(ioPin, onState);
    ledState = LED_ON;
  }

  /**
   * Controls LED with external variable/condition
   *
   * @param ledState New state of LED (false or true)
   */
  public void putState(boolean ledState) {
    if (ledState)
      on();
    else
      off();
  }

  /**
   * Inverts state of LED
   */
  public void invert() {
    putState(!ledState);
  }

  /**
   * Returns LED status
   *
   * return LED status (LED_OFF, LED_ON)
   */
  public boolean getState() {
    return ledState;
  }
}
```

Program Listing 86.2

```
// LED object demonstration
//
// by Jon Williams
// jwilliams@parallaxinc.com
//
// 03 MAY 2002

import stamp.core.*;
import stamp.peripheral.io.LED;

public class HelloLEDs {

  public static void main() {

    // create LEDs
    LED green = new LED(CPU.pin0, true);
    LED yellow = new LED(CPU.pin1, true);
    LED red = new LED(CPU.pin2, true);

    // create timers for LEDs
    Timer greenTimer = new Timer();
    Timer yellowTimer = new Timer();
    Timer redTimer = new Timer();

    // demonstrate LED basics
    green.on();
    CPU.delay(5000);
    yellow.putState(true);
    CPU.delay(5000);
    red.putState(yellow.getState());
    CPU.delay(5000);

    green.off();
    CPU.delay(5000);
    yellow.putState(false);
    CPU.delay(5000);
    red.invert();
    CPU.delay(5000);

    // start the timers
    greenTimer.mark();
    yellowTimer.mark();
    redTimer.mark();

    // flash LEDs at different rates
    while (true) {
      if (greenTimer.timeout(250)) {
        greenTimer.mark();
        green.invert();
      }
      if (yellowTimer.timeout(333)) {
        yellowTimer.mark();
        yellow.invert();
      }
      if (redTimer.timeout(1000)) {
        redTimer.mark();
        red.invert();
      }
    }
  }
}
```