§

Computer Architecture

Introduction

Robots are in widespread use in the car, medical, and manufacturing industries, in all manner of exploration vehicles, and, of course, in many science fiction films. The word 'robot' first appeared in a Czechoslovakian satirical play, Rossum's Universal Robots, by Karel Capek in 1920. Robots in this play tended to be human-like. From this point onward, it seemed that many science fiction stories involved these robots trying to fit into society and make sense out of human emotions. This changed when General Motors installed the first robots in its manufacturing plant in 1961. These automated machines presented an entirely different image from the "human form" robots of science fiction.

Building and programming a robot is a combination of mechanics, electronics, and problem-solving. What you are about to learn while doing the activities and projects in this text will be relevant to real-world applications that use robotic control, the only differences being the size and sophistication. The mechanical principles, example program listings, and circuits you will use are similar to very common elements in industrial applications.

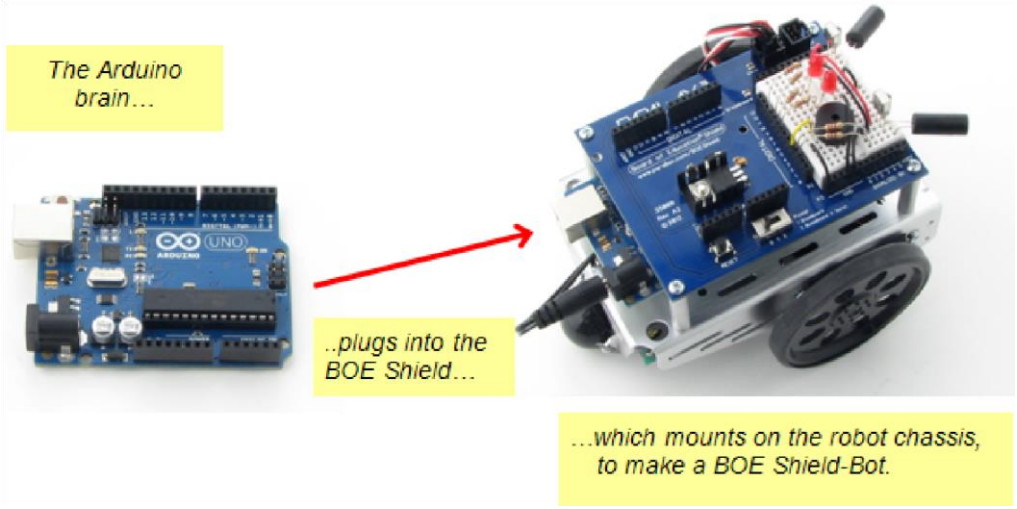
The activities and projects you will complete in these labs start with an introduction to your Bot's brain, the Arduino Uno. Then, you will build, test, and calibrate the Shield-Bot. Next, you will learn to program the Shield-Bot for basic manoeuvres. After that, you'll be ready to add different kinds of sensors, and write sketches to make the Shield-Bot sense its environment and respond on its own.

§

Lab 1

Your Shield-Bot's Brain

The BOE Shield mounts on a metal chassis with servo motors and wheels. Your Arduino module—the programmable brain—plugs in underneath the Shield.



You will use the Arduino Development Environment software to write programs (called sketches) that will make your BOE Shield-Bot do those four essential robotic tasks. Arduino sketches are written by adding C and C++ programming language statements to a beginner-friendly template.

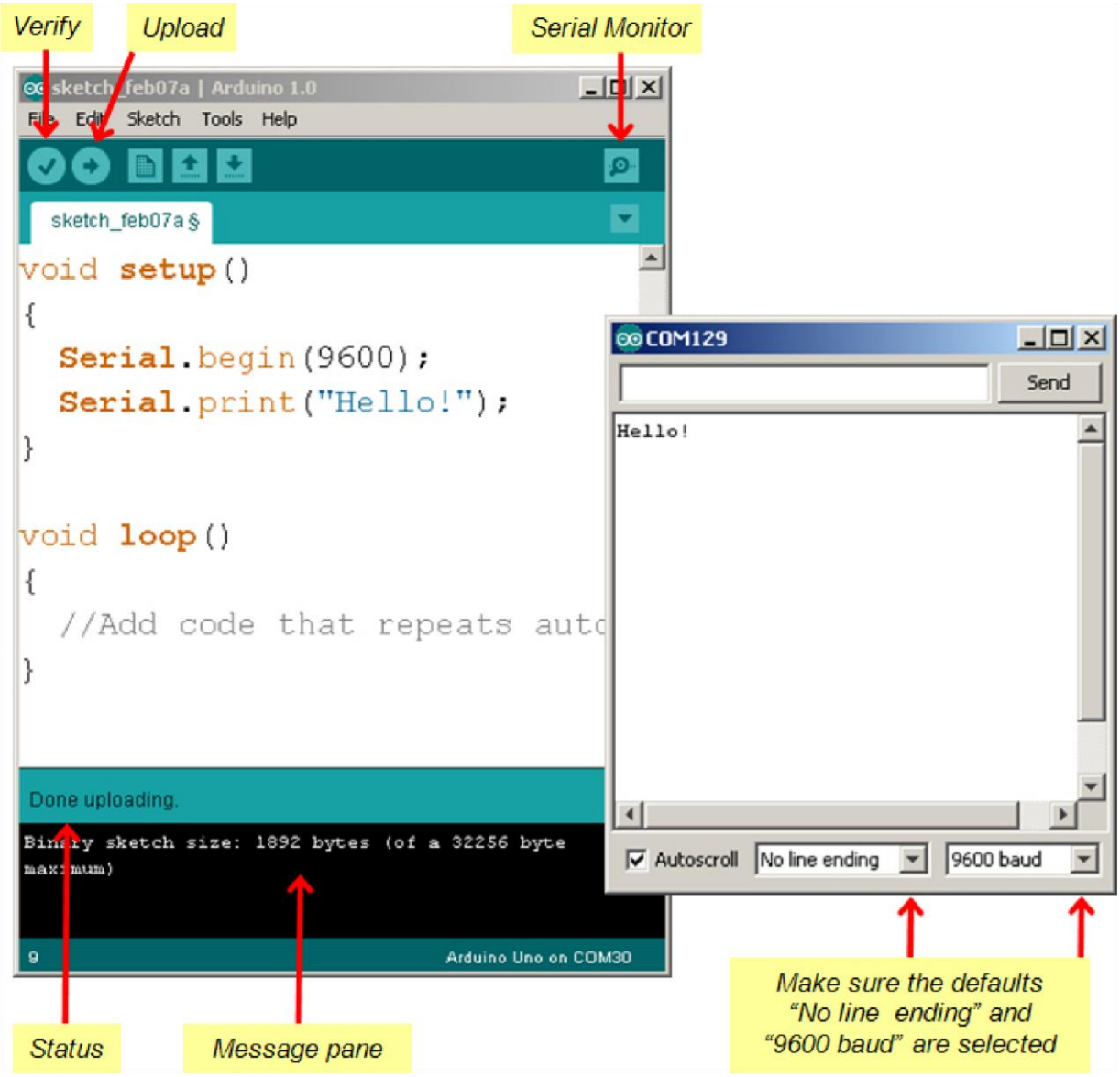
Here is a screen capture of the Arduino Development Environment edit pane on the left, containing a simple sketch that sends a “Hello!” message to the Serial Monitor window on the right.

- Open your Arduino software and carefully type in the code:

```
void setup()

{
  Serial.begin(9600);
  Serial.print("Hello!");
}

void loop()
{
  //Add code that repeats automatically here.
}
```

- Be sure you have capitalized “Serial” both times, or the sketch won’t work.
- Also, notice in the figure that the sketch uses parentheses `()` and curly braces `{}`. Be sure to use the right ones in the right places!

§



- ✓ Click the Verify button to make sure your code doesn't have any typing errors.
- ✓ Look for the "Binary sketch size" text in the message pane.
  - If it's there, your code compiled and is ready to upload to the Arduino.
  - If there's a list of errors instead, it's trying to tell you it can't compile your code. So, find the typing mistake and fix it!
- ✓ Click the Upload button. The status line under your code will display "Compiling sketch…," "Uploading…," and then "Done uploading."
- ✓ After the sketch is done uploading, click the Serial Monitor button.
- ✓ If the Hello message doesn't display as soon as the Serial Monitor window opens, check for the "9600 baud" setting in the lower right corner of the monitor.
- ✓ Use File → Save to save your sketch. Give it the name HelloMessage.

§

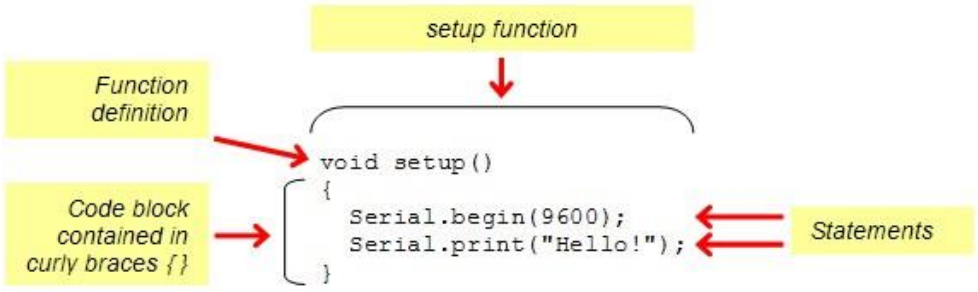
How the Hello Sketch Code Works

A function is a container for statements (lines of code) that tell the Arduino to do certain jobs. The Arduino language has two built-in functions: `setup` and `loop`. The `setup` function is shown below. The Arduino executes the statements you put between the `setup` function's curly braces, but only once at the beginning of the program.

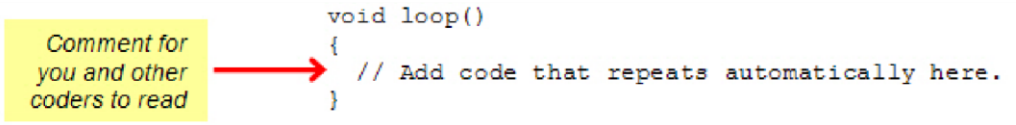
In this example, both statements are function calls to functions in the Arduino's built-in Serial pre-written code library: `Serial.begin(speed)` and `Serial.print(val)`. Here, speed and val are parameters, each describing a value that its function needs passed to it to do its job. The sketch provides these values inside parentheses in each function call.

`Serial.begin(9600);` passes the value 9600 to the speed parameter. This tells the Arduino to get ready to exchange messages with the Serial Monitor at a data rate of 9600 bits per second. That's 9600 binary ones or zeros per second, and is commonly called a baud rate.

`Serial.print(val);` passes the message "Hello!" to the val parameter. This tells the Arduino to send a series of binary ones and zeros to the Serial Monitor. The monitor decodes and displays that serial bitstream as the "Hello!" message.



After the `setup` function is done, the Arduino automatically skips to the `loop` function and starts doing what the statements in its curly braces tell it to do. Any statements in `loop` will be repeated over and over again, indefinitely. Since all this sketch is supposed to do is print one "Hello!" message, the `loop` function doesn't have any actual commands. There's just a notation for other programmers to read, called a comment. Anything to the right of `//` on a given line is for programmers to read, not for the Arduino software's compiler. (A compiler takes your sketch code and converts it into numbers—a microcontroller's native language.)



What is `void`? Why do these functions end in `()`? The first line of a function is its definition, and it has three parts: return type, name, and parameter list. For example, in the function `void setup()` the return type is void, the name is `setup`, and the parameter list is empty – there's nothing inside the parentheses ().
Void means 'nothing'—when another function calls `setup` or `loop`, these functions would not return a value. An empty parameter list means that these functions do not need to receive any values when they are called to do their jobs.

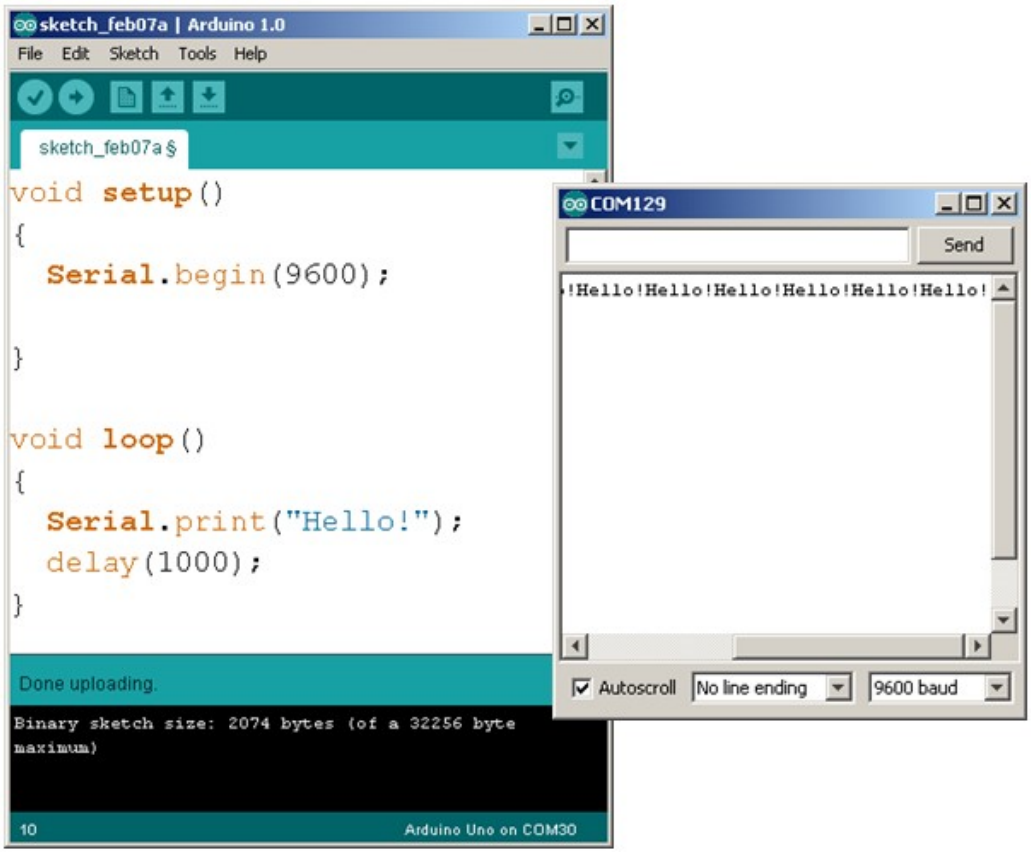
§

Modify the Sketch to Repeat

Microcontroller programs generally run in a loop, meaning that one or more statements are repeated over and over again. Remember that the `loop` function automatically repeats any code in its block (the statements in between its curly braces). Let's try moving `Serial.print("Hello!");` to the `loop` function. To slow down the rate at which the messages repeat, let's also add a pause with the built-in `delay(ms)` function.

- ✓ Save HelloMessage as HelloRepeated.
- ✓ Move `Serial.print("Hello!");` from `setup` to the `loop` function.
- ✓ Add `delay(1000);` on the next line.
- ✓ Compare your changes to the figure below and verify that they are correct.
- ✓ Upload the sketch to the Arduino and then open the Serial Monitor again.

The added line `delay(1000)` passes the value 1000 to the `delay` function's ms parameter. It's requesting a delay of 1000 milliseconds. 1 ms is 1/1000 of a second. So, `delay(1000)` makes the sketch wait for 1000/1000 = 1 second before letting it move on to the next line of code.



§

Hello Messages on New Lines

How about having each "Hello!" message on a new line? That would make the messages scroll down the Serial Monitor, instead of across it. All you have to do is change `print` to `println`, which is short for 'print line.'

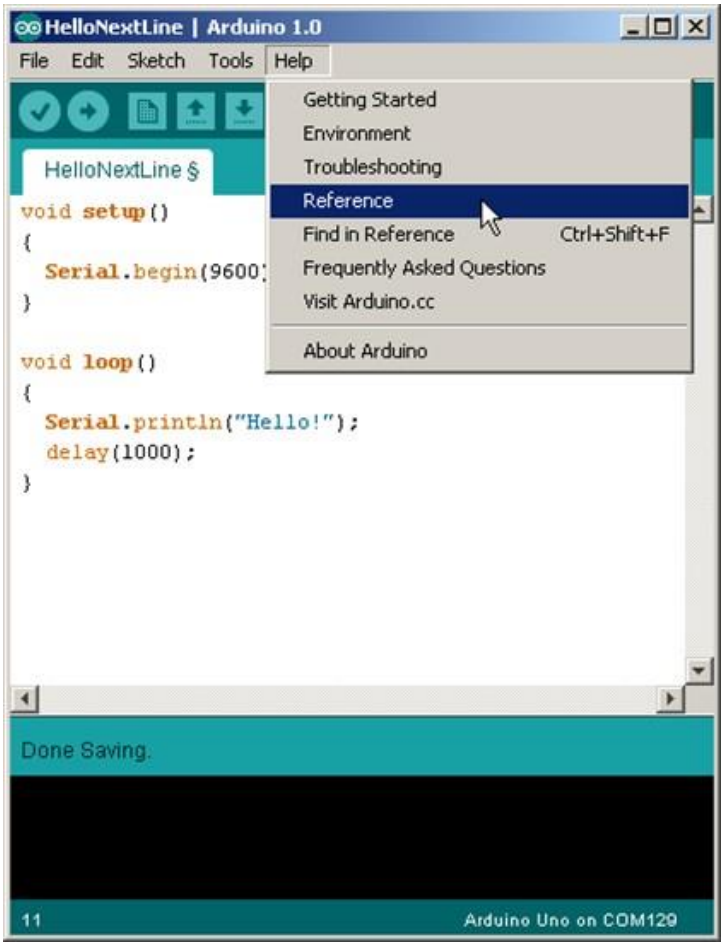
- Change `Serial.print("Hello!")` to `Serial.println("Hello!")`.
- Upload the modified sketch and watch it print each "Hello!" message on a new line.

§

Open the Arduino Reference

Still have questions? Try the Arduino Language Reference. It's a set of pages with links you can follow to learn more about setup, loop, print, println, delay, and lots of other functions you can use in your sketches.

- Click Help and Select Reference.



A reference like the one below should open into a browser.

- Try looking up whichever term you might have a question about. You'll find `setup`, `loop`, and `delay` on the main reference page.
- If you're looking for links to `print` or `println`, you'll have to find and follow the `Serial` link first.

§

Store and Retrieve Values

Variables are names you can create for storing, retrieving, and using values in the Arduino microcontroller's memory. Here are three example variable declarations from the next sketch:

```
int a = 42;   char c = 'm';
float root2 = sqrt(2.0);
```

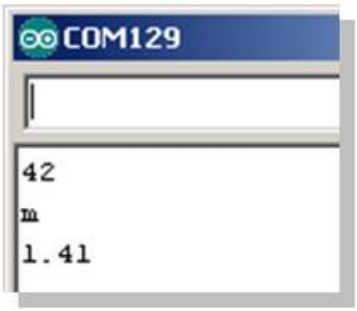
The declaration `int a = 42` creates a variable named `a`. The `int` part tells the Arduino software what type of variable it's dealing with. The `int` type can store integer values ranging from -32,768 to 32,767. The declaration also assigns `a` an initial value of 42. (The initial value is optional, you could instead just declare `int a`, and then later assign the value 42 to a with `a = 42`.)

Next, `char c = 'm'` declares a variable named `c` of the type `char` (which is for storing characters) and then assigns it the value 'm'.

Then, `float root2 = sqrt(2.0)` declares a variable named `root2`. The variable type is `float`, which can hold decimal values. Here, `root2` is initialized to the floating-point representation of the square root of two: `sqrt(2.0)`.

Now that your code has stored values to memory, how can it retrieve and use them? One way is to simply pass each variable to a function's parameter. Here are three examples, where the `Serial.println(val)` function displays the value of the variable inside the parentheses.

```
Serial.println(a);
Serial.println(c);
Serial.println(root2);
```



One nice thing about variable types is that `Serial.println` recognizes each type and displays it correctly in the serial monitor. (Also, the C++ compiler in the Arduino software requires all declared variables to have a type, so you can't leave it out.)

Example Sketch – StoreRetrieveLocal

- Use File → New to create a new sketch, and save it as StoreRetrieveLocal.
- Enter or copy the code below into the Arduino editor.
- Save the file, then upload it to the Arduino.
- Open the Serial Monitor and verify that the values display correctly.

```
// Robotics with the BOE Shield - StoreRetrieveLocal

void setup()
{
  Serial.begin(9600);

int a = 42;   char c = 'm';
float root2 = sqrt(2.0);
```

§

```
  Serial.println(a);
  Serial.println(c);
  Serial.println(root2);
}

void loop()
{
  // Empty, no repeating code.
}
```

ASCII stands for American Standard Code for Information Exchange.

1

It's a common code system for representing computer keys and characters in displays. For example, both the Arduino and the Serial Monitor use the ASCII code 109 for the letter m. The declaration char c = 'm' makes the Arduino store the number 109 in the c variable. Serial.println(c) makes the Arduino send the number 109 to the Serial Monitor. When the Serial Monitor receives that 109, it automatically displays the letter m. View ASCII codes 0–127 [15].

See that 'm' really is 109

There are two ways to prove that the ASCII code for 'm' really is 109. First, instead of declaring `char c = 'm'`, you could use `byte c = 'm'`. Then, the `println` function will print the `byte` variable's decimal value instead of the character it represents. Or, you could leave the `char c` declaration alone and instead use `Serial.println(c, DEC)` to display the decimal value `c` stores.

- ✓ Try both approaches.

So, do you think the letters l, m, n, o, and p would be represented by the ASCII codes 108, 109, 110, 110, 111, and 112?

- ✓ Modify your sketch to find out the decimal ASCII codes for l, m, n, o, p.

Global vs.Local Variables

So far, we've declared variables inside a function block (inside the function's curly braces), which means they are local variables. Only the function declaring a local variable can see or modify it. Also, a local variable only exists while the function that declares it is using it. After that, it gets returned to unallocated memory so that another function (like `loop`) could use that memory for a different local variable.

If your sketch has to give more than one function access to a variable's value, you can use global variables. To make a variable global, just declare it outside of any function, preferably before the `setup` function. Then, all functions in the sketch will be able to modify or retrieve its value. The next example sketch declares global variables and assigns values to them from within a function.

Example Sketch – StoreRetrieveGlobal

This example sketch declares a, c, and root2 as global variables (instead of local). Now that they are global, both the setup and loop functions can access them.

Modify your sketch so that it matches the one below.

Save the file as StoreRetrieveGlobal, then upload it to the Arduino.

§

Open the Serial Monitor and verify that the correct values are displayed repeatedly by the loop function.

```
int a; char c; float root2;    void setup()
{
 Serial.begin(9600);
 a = 42;  c = 'm';
  root2 = sqrt(2.0);
  }

  void loop()
  {
    Serial.println(a);
    Serial.println(c);
  Serial.println(root2);    delay(1000);
  }
```

Your Turn – More Variable Types

There are lots more data types than just `int`, `char`, `float`, and `byte`.

- ✓ Open the Arduino Language Reference, and check out the Data Types list.
- ✓ Follow the `float` link and learn more about this data type.
- ✓ The `long` data type will be used in a later chapter; open both the `long` and `int` sections. How are they similar? How are they different?

Solve Maths Problems

Arithmetic operators are useful for doing calculations in your sketch. In this activity, we'll focus on the basics: assignment (=), addition (+), subtraction (-), multiplication (*), division(/), and modulus (%, the remainder of a division calculation).

✓ Open up the Arduino Language Reference, and take a look at the list of Arithmetic Operators.

The next example sketch, SimpleMath, adds the variables `a` and `b` together and stores the result in `c`. It also displays the result in the Serial Monitor.

Notice that `c` is now declared as an `int`, not a `char` variable type. Another point, `int c = a + b` uses the assignment operator (=) to copy the result of the addition operation that adds `a` to `b`. The figure below shows the expected result of 89 + 42 = 131 in the Serial Monitor.

9

- Enter, save, and upload SimpleMath to your Arduino.
- Check the result in the Serial Monitor. Is it correct?

```
// Robotics with the BOE Shield - SimpleMath

void setup()
{
  Serial.begin(9600);

  int a = 89;
int b = 42;
  int c = a + b;
```

3

```
  Serial.print("a + b = ");
  Serial.println(c);
}

void loop()
{
  // Empty, no repeating code.
}
```

Fit your variables to the result values you need to store.
This will use less memory so you can write larger sketches that will execute more efficiently.

If you need to work with decimal point values, use `float`.

If you are using integer values (counting numbers), choose `byte`, `int`, or `long`.

If your results will always be an unsigned number from 0 to 255, use `byte`.

If your results will not exceed –32,768 to 32,767, an `int` variable can store your value.

If you need a larger range of values, try a `long` variable instead. It can store values from -2,147,483,648 to 2,147,483,647.

Your Turn – Experiment with Other Arithmetic Operators

You still have –, *, /, and % to try out!

- Replace the addition (+) operator with the subtraction (–) operator and re-upload the sketch to the Arduino. (You'll want to replace both instances of `+` in the sketch.) Upload and verify the result in the
- Serial Monitor.
- Repeat for the multiplication (`*`), division (`/`) and modulus (`%`) operators.

§

Floating Point Maths

Imagine your BOE Shield-Bot will enter a contest where you have to make it travel in a circle, but the radius of the circle will only be announced a few minutes before the contest. In this situation, you'll have an advantage if your code can calculate the circumference of the circle. The circumference is 2 × π × r, where r is the circle's radius and π ≈ 3.14159. This calculation would be a lot easier to do with floating point math.

Here is a snippet of code that gets the job done. Notice that it uses PI instead of 3.14159. `PI` is a built-in C language constant (a named value that does not change throughout the sketch). Also notice that all the values have decimal points. That makes them all floating-point values.

```
float r = 0.75;

float c= 2.0 * PI * r;
```

Example Sketch - Circumference

- ✓ Enter the Circumference sketch into the Arduino editor and save it.

4

- ✓ Make sure to use the values 0.75 and 2.0. Do not try to use 2 instead of 2.0.
- ✓ Upload your sketch to the Arduino and check the results with the Serial Monitor.

```
// Robotics with the BOE Shield - Circumference

void setup()
{
  Serial.begin(9600);

  float r = 0.75;   float c
= 2.0 * PI * r;


  Serial.print("circumference = ");
  Serial.println(c); }

void loop()
{
  // Empty, no repeating code.
}
```

Your Turn – Circle Area

The area of a circle is a = π × r2. Hint: r2 can be expressed as simply r × r.

- ✓ Save your sketch as CircumferenceArea.
- ✓ Add another `float` variable to store the result of an area calculation, and code to display it. Run the sketch, and test the answer against a calculator.

§

Make Decisions

Your BOE Shield-Bot will need to make a lot of navigation decisions based on sensor inputs. Here is a simple sketch that demonstrates decision-making. It compares the value of `a` to `b`, and sends a message to tell you whether or not `a` is greater than `b`, with an `if…else` statement.

If the condition `(a > b)` is true, it executes the `if` statement's code block: `Serial.print("a is greater than b")`. If `a` is not greater than `b`, it executes else code block instead: `Serial.print("a is not greater than b")`.

- Enter the code into the Arduino editor, save it, and upload it to the Arduino.
- Open the Serial Monitor and test to make sure you got the right message.
- Try swapping the values for a and b.
- Re-upload the sketch and verify that it printed the other message.

```
// Robotics with the BOE Shield - SimpleDecisions

void setup()
{
  Serial.begin(9600);

  int a = 89;
int b = 42;

  if(a > b)
```



```
   {
     Serial.print("a is greater than b");
   }
 else
   {
     Serial.print("a is not greater than b");
   }
 }

 void loop()
 {
   // Empty, no repeating code. }
```

More Decisions with if... else if

Maybe you only need a message when `a` is greater than `b`. If that's the case, you could cut out the `else` statement and its code block. So, all your `setup` function would have is the one `if` statement, like this:

```
void setup()
{
  Serial.begin(9600);

  int a = 89;

  int b = 42;

  if(a > b)
```

§

```
  {
    Serial.print("a is greater than b");
  }
}
```

Maybe your sketch needs to monitor for three conditions: greater than, less than, or equal. Then, you could use an `if…else if…else` statement.

```
if(a > b)
  {
    Serial.print("a is greater than b");
  }
  else if(a < b)
  {
    Serial.print("b is greater than a");
  }
else
  {
    Serial.print("a is equal to b");
  }
```

A sketch can also have multiple conditions with the Arduino's boolean operators, such as && and ||. The && operator means AND; the || operator means OR. For example, this statement's block will execute only if `a` is greater than 50 AND `b` is less than 50:

```
 if((a > 50) && (b < 50))
  {
    Serial.print("Values in normal range");    }
```

Another example: this one prints the warning message if `a` is greater than 100 OR `b` is less than zero.

6

```
if((a > 100) || (b < 0))
  {
    Serial.print("Danger Will Robinson!");
  }
```

One last example: if you want to make a comparison to find out if two values are equal, you have to use two equal signs next to each other ==.

```
if(a == b)
  {
    Serial.print("a and b are equal");
  }
```

✓ Try these variations in a sketch.

You can chain more `else if` statements after `if`.
The example in this activity only uses one `else if`, but you could use more.

The rest of the statement gets left behind after it finds a true condition. If the `if` statement turns out to be true, its code block gets executed and the rest of the chain of `else if`s gets passed by.

§

Count and Control Repetitions

Many robotic tasks involve repeating an action over and over again. Next, we'll look at two options for repeating code: the `for` loop and `while` loop. The `for` loop is commonly used for repeating a block of code a certain number of times. The `while` loop is used to keep repeating a block of code as long as a condition is true.

A for Loop is for Counting

A `for` loop is typically used to make the statements in a code block repeat a certain number of times. For example, your BOE Shield-Bot will use five different values to make a sensor detect distance, so it needs to repeat a certain code block five times. For this task, we use a `for` loop. Here is an example that uses a `for` loop to count from 1 to 10 and display the values in the Serial Monitor.

7

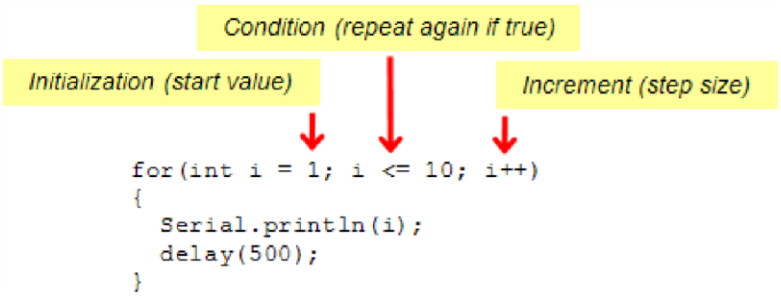
- Enter, save, and upload CountToTen.
- Open the Serial Monitor and verify that it counted from one to ten.

```
// Robotics with the BOE Shield - CountToTen

void setup()
{
  Serial.begin(9600);


  for(int i = 1; i <= 10; i++)
  {
     Serial.println(i);
     delay(500);
  }
  Serial.println("All done!"); }

void loop()
{
  // Empty, no repeating code. }
```

§

How the for Loop Works

The figure below shows the `for` loop from the last example sketch, CountTenTimes. It labels the three elements in the **`for`** loop's parentheses that control how it counts.



- Initialization: the starting value for counting. It's common to declare a local variable for the job as we did here with **`int i = 1;`** naming it **`i`** for 'index.'
- Condition: what the **`for`** loop checks between each repetition to make sure the condition is still true. If it's true, the loop repeats again. If not, it allows the code to move on to the next statement that follows the **`for`** loop's code block. In this case, the conidition is "if `i` is less than or equal to 10."
- Increment: how to change the value of `i` for the next time through the loop. The expression `i++` is equivalent to `i = i + 1`. It makes a nice shorthand approach for adding 1 to a variable. Notice that `++` comes after `i`, meaning "use `i` as-is this time through the function, and then increment it afterward." This is the post increment use of the operator.

The first time though the loop, the value of `i` starts at 1. So, `Serial.println(i)` displays the value 1 in the Serial Monitor. The next time through the loop, `i++ has made` the value of `i` increase by 1. After a delay (so you can watch the individual values appear in the Serial Monitor), the for statement checks to make sure the condition `i <= 10` is still true. Since `i` now stores 2, it is true since 2 is less than 10, so it allows the code block to repeat again. This keeps repeating, but when `i` gets to 11, it does not execute the code block because it's not true according to the `i <= 10` condition.

Adjust Initialization, Condition, and Increment

8

As mentioned earlier, i++ uses the ++ increment operator to add 1 to the i variable each time through the for loop. There are also compound operators for decrement --, and compound arithmetic operators like +=, -=, *= and /=. For example, the += operator can be used to write i = i + 1000 like this: i+=1000.

- ✓ Save your sketch, then save it as CountHigherInSteps.
- ✓ Replace the `for` statement in the sketch with this:

```
for(int i = 5000; i <= 15000; i+=1000)
```

- ✓ Upload the modified sketch and watch the output in the Serial Monitor.

A Loop that Repeats While a Condition is True

§

In later chapters, you'll use a `while` loop to keep repeating things while a sensor returns a certain value. We don't have any sensors connected right now, so let's just try counting to ten with a `while` loop:

```
int i = 0;
while(i < 10)
  {
    i = i + 1;
Serial.println(i);     delay(500);
  }
```

Want to condense your code a little? You can use the increment operator (++) to increase the value of `i` inside the `Serial.println` statement. Notice that `++` is on the left side of the `i` variable in the example below. When `++` is on the left of a variable, it adds 1 to the value of `i` before the `println` function executes. If you put `++` to the right, it would add 1 after println executes, so the display would start at zero.

```
 int i = 0;
while(i < 10)
  {
    Serial.println(++i);
delay(500);
  }
```

The `loop` function, which must be in every Arduino sketch, repeats indefinitely. Another way to make a block of statements repeat indefinitely in a loop is like this:

```
 int i = 0;
while(true)
  {
    Serial.println(++i);
delay(500);
```

So why does this work? A `while` loop keeps repeating as long as what is in its parentheses evaluates as true. The word 'true' is actually a pre-defined constant, so `while(true)` is always true, and will keep the `while` loop looping. Can you guess what `while(false)` would do?

- Try these three different while loops in place of the for loop in the CountToTen sketch.
- Also try one instance using `Serial.println(++i)`, and watch what happens in the Serial Monitor.
- Try `while(true)` and `while(false)` also.

9

Constants and Comments

The next sketch, CountToTenDocumented, is different from CountToTen in several ways. First, it has a block comment at the top. A block comment starts with `/*` and ends with `*/`, and you can write as many lines of notes in between as you want. Also, each line of code has a line comment (starting with `//`) to its right, explaining what the code does.

Last, two `const int` (constants that are integers) are declared at the beginning of the sketch, giving the names `startVal`, `endVal`, and `baudRate` to the values 1, 10, and 9600. Then, the sketch uses these names wherever it requires these values.

```
/*
  Robotics with the BOE Shield – CountToTenDocumented
  This sketch displays an up-count from 1 to 10 in the Serial Monitor
  */

 const int startVal = 1;                        // Starting value for
 counting const int endVal = 10;                         // Ending value
 for counting const int baudRate = 9600;                  // For
 setting baud rate

 void setup()                                   // Built in initialization
 block
 {
   Serial.begin(baudRate);                      // Set data rate to baudRate


   for(int i = startVal; i <= endVal; i++)      // Count from startVal to
 endVal
   {
     Serial.println(i);                         // Display i in Serial
 Monitor     delay(500);                               // Pause 0.5 s
 between values
   }
   Serial.println("All done!");                 // Display message when done
 }

 void loop()                                    // Main loop auto-repeats
 {
 // Empty, no repeating code.
}
```

Documenting Code

Documenting code is the process of writing notes about what each part of the program does. You can help make your code self-documenting by picking variable and constant names that help make the program more self-explanatory. If you are thinking about working in a field that involves programming, it's a good habit to start now. Why?

- Folks who write code for a living, like software developers and robotics programmers, are usually under orders to document their code.
- Other people might need to make updates to your code or use it for another project, and they need to understand what the code does.

Documented code can save you lots of time trying to remember what your code does, and how it does it, after you haven't looked at it for a long time.

In addition to making your code easier to read, constants allow you to adjust an often-used value quickly and accurately by updating a single constant declaration. Trying to find and update each instance of an unnamed value by hand is an easy way to create bugs in your sketch.

- Read through the sketch to see how constants and comments are used.
- Open up the SimpleDecisions sketch and document it, using the sketch CountToTenDocumented as an example.

2

Add a detailed description of the sketch, enclosing it with the title in a block comment.
Use `const` declarations for the values of 89 and 42.
Use the names from your `const` declarations instead of 89 and 42 in the setup function.
Add line comments to the right of each line.

Ensure that you individually upload your completed code to Moodle each week.