Arduino Programming

Commonly used functions

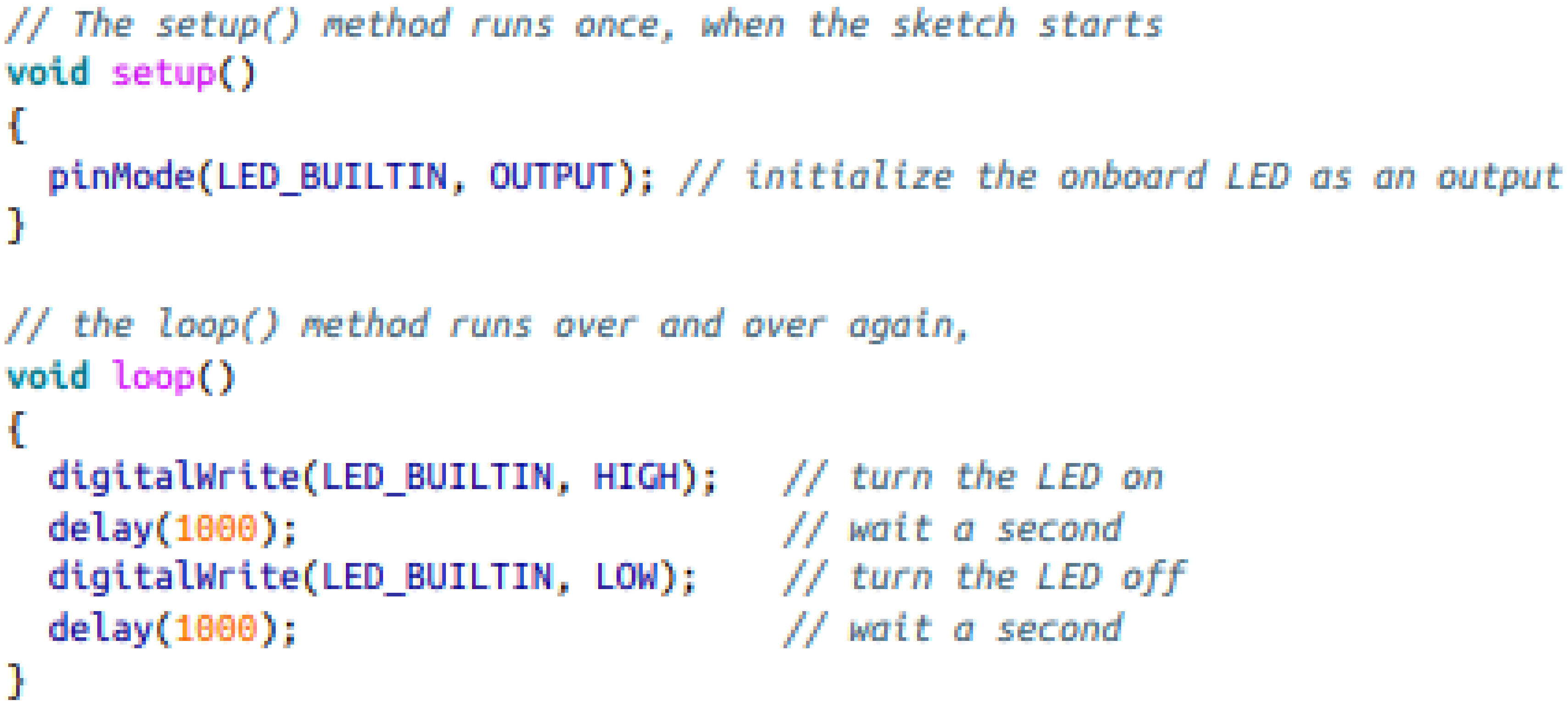
- Serial.println(value)
  - ✓Prints the value to the Arduino IDE's Serial Monitor so you can view Arduino's output on your computer
- pinMode(pin, mode)
  - ✓Configures a digital pin to read (input) or write (output) a digital value
- digitalRead(pin)
  - ✓Reads a digital value (HIGH or LOW) on a pin set for input
- digitalWrite(pin, value)
  - ✓Writes the digital value (HIGH or LOW) to a pin set for output

A Typical Arduino Sketch

- Programs for Arduino are usually referred to as sketches.
- Sketches contain code—the instructions the board will carry out
- Code that needs to run only once (such as to set up the board for your application) must be placed in the setup function.
- Code to be run continuously after the initial setup has finished goes into the loop function

A Typical Arduino Sketch (Continue)

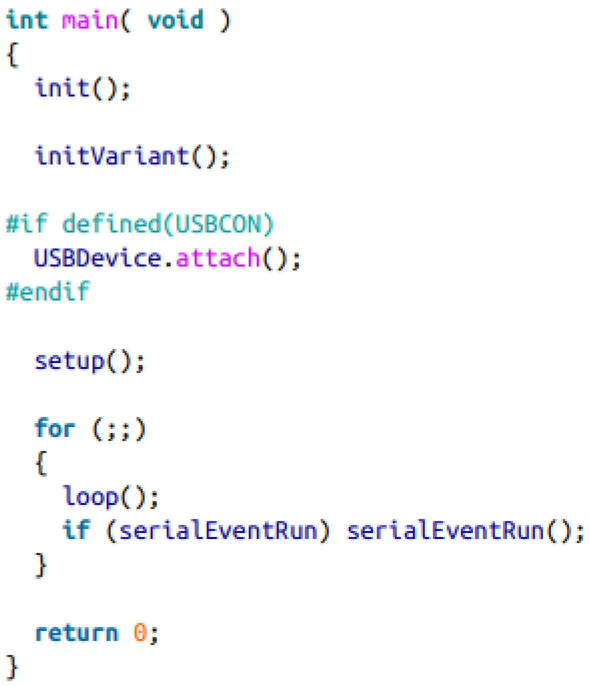
- Programs an Arduino to continually flash an LED light.



```
// The setup() method runs once, when the sketch starts
void setup()
{
  pinMode(LED_BUILTIN, OUTPUT); // initialize the onboard LED as an output
}

// the loop() method runs over and over again,
void loop()
{
  digitalWrite(LED_BUILTIN, HIGH);   // turn the LED on
  delay(1000);                       // wait a second
  digitalWrite(LED_BUILTIN, LOW);    // turn the LED off
  delay(1000);                       // wait a second
}
```

A Typical Arduino Sketch (Continue)

- The main function looks like for 8-bit boards



```
int main( void )
{
  init();

  initVariant();

#if defined(USBCON)
  USBDevice.attach();
#endif

  setup();

  for (;;)
  {
    loop();
    if (serialEventRun) serialEventRun();
  }

  return 0;
}
```

A Typical Arduino Sketch (Continue)

- The first thing that happens is a call to an init() function that initializes the Arduino hardware.
- After that, initVariant() gets called. This is a rarely used hook to give makers of Arduino-compatible boards a way to invoke their own custom initialization routines.
- If the microcontroller on the board has dedicated USB hardware, main will prepare (attach) it for use.
- Next, your sketch’s setup() function is called.
- Finally, your loop() function is called over and over.
- Because the for loop never terminates, the return statement is never executed.

Using Simple Primitive Types (Variables)

- Arduino has different types of variables to efficiently represent values. You want to know how to select and use these Arduino data types.

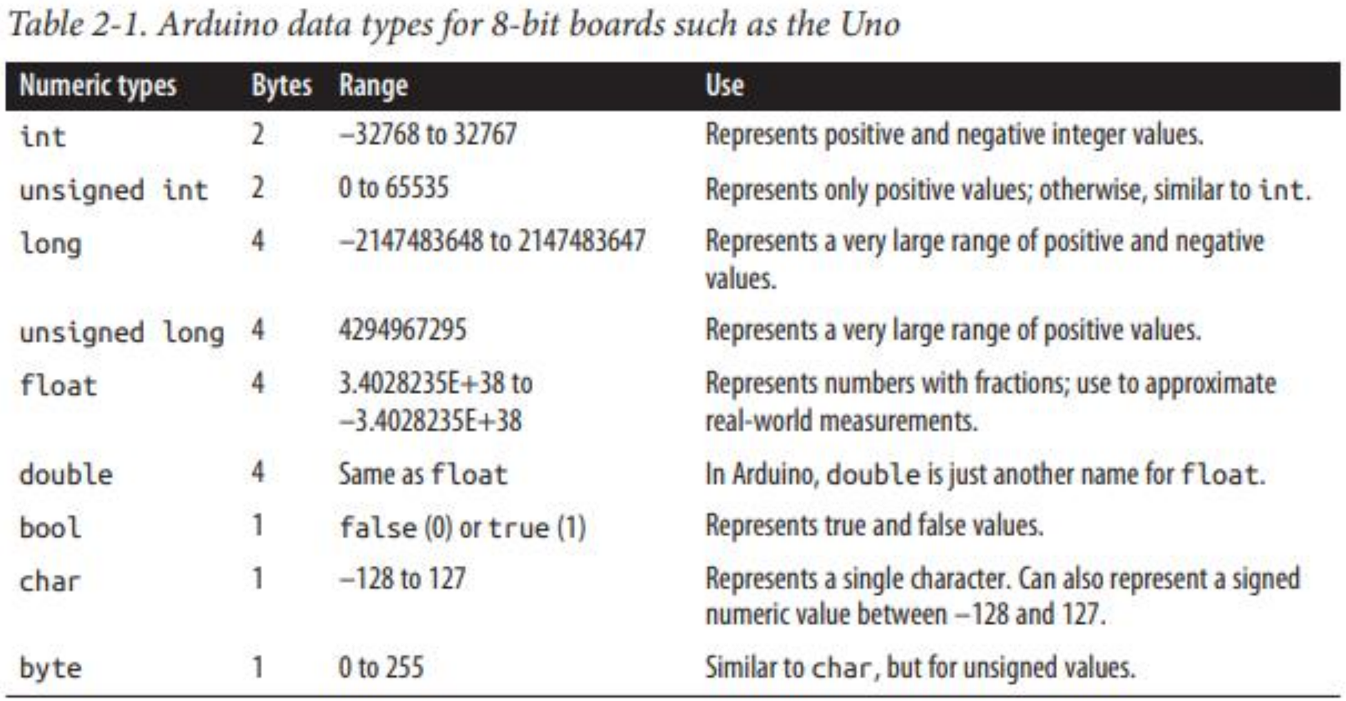
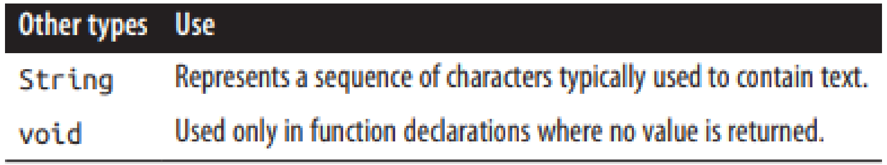


Table 2-1. Arduino data types for 8-bit boards such as the Uno

Numeric types	Bytes	Range	Use
`int`	2	–32768 to 32767	Represents positive and negative integer values.
`unsigned int`	2	0 to 65535	Represents only positive values; otherwise, similar to `int`.
`long`	4	–2147483648 to 2147483647	Represents a very large range of positive and negative values.
`unsigned long`	4	4294967295	Represents a very large range of positive values.
`float`	4	3.4028235E+38 to –3.4028235E+38	Represents numbers with fractions; use to approximate real-world measurements.
`double`	4	Same as `float`	In Arduino, `double` is just another name for `float`.
`bool`	1	`false` (0) or `true` (1)	Represents true and false values.
`char`	1	–128 to 127	Represents a single character. Can also represent a signed numeric value between –128 and 127.
`byte`	1	0 to 255	Similar to `char`, but for unsigned values.



Other types	Use
`String`	Represents a sequence of characters typically used to contain text.
`void`	Used only in function declarations where no value is returned.

Using Simple Primitive Types (Variables)

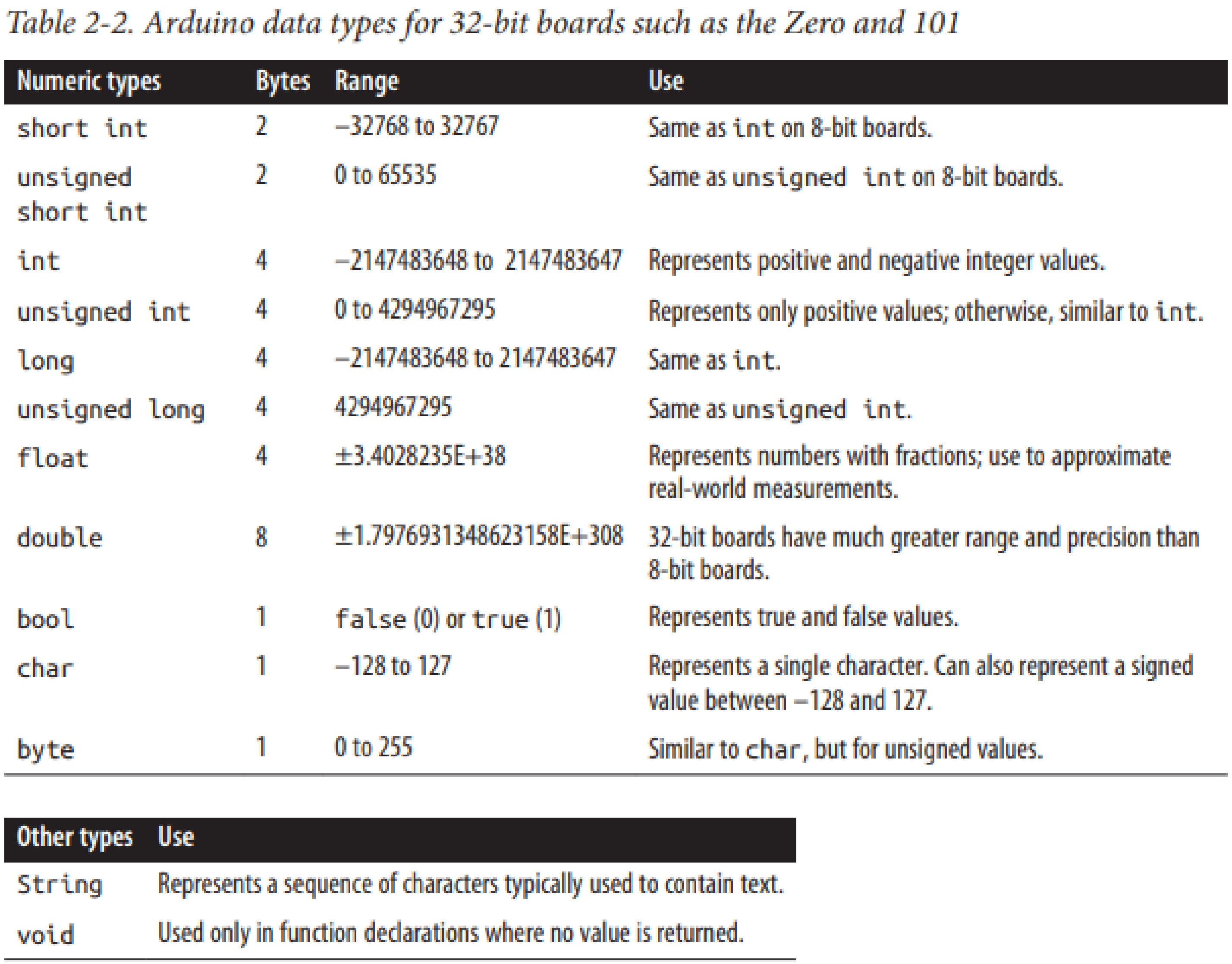


Table 2-2. Arduino data types for 32-bit boards such as the Zero and 101

Numeric types	Bytes	Range	Use
`short int`	2	–32768 to 32767	Same as `int` on 8-bit boards.
`unsigned short int`	2	0 to 65535	Same as `unsigned int` on 8-bit boards.
`int`	4	–2147483648 to 2147483647	Represents positive and negative integer values.
`unsigned int`	4	0 to 4294967295	Represents only positive values; otherwise, similar to `int`.
`long`	4	–2147483648 to 2147483647	Same as `int`.
`unsigned long`	4	4294967295	Same as `unsigned int`.
`float`	4	±3.4028235E+38	Represents numbers with fractions; use to approximate real-world measurements.
`double`	8	±1.7976931348623158E+308	32-bit boards have much greater range and precision than 8-bit boards.
`bool`	1	`false` (0) or `true` (1)	Represents true and false values.
`char`	1	–128 to 127	Represents a single character. Can also represent a signed value between –128 and 127.
`byte`	1	0 to 255	Similar to `char`, but for unsigned values.

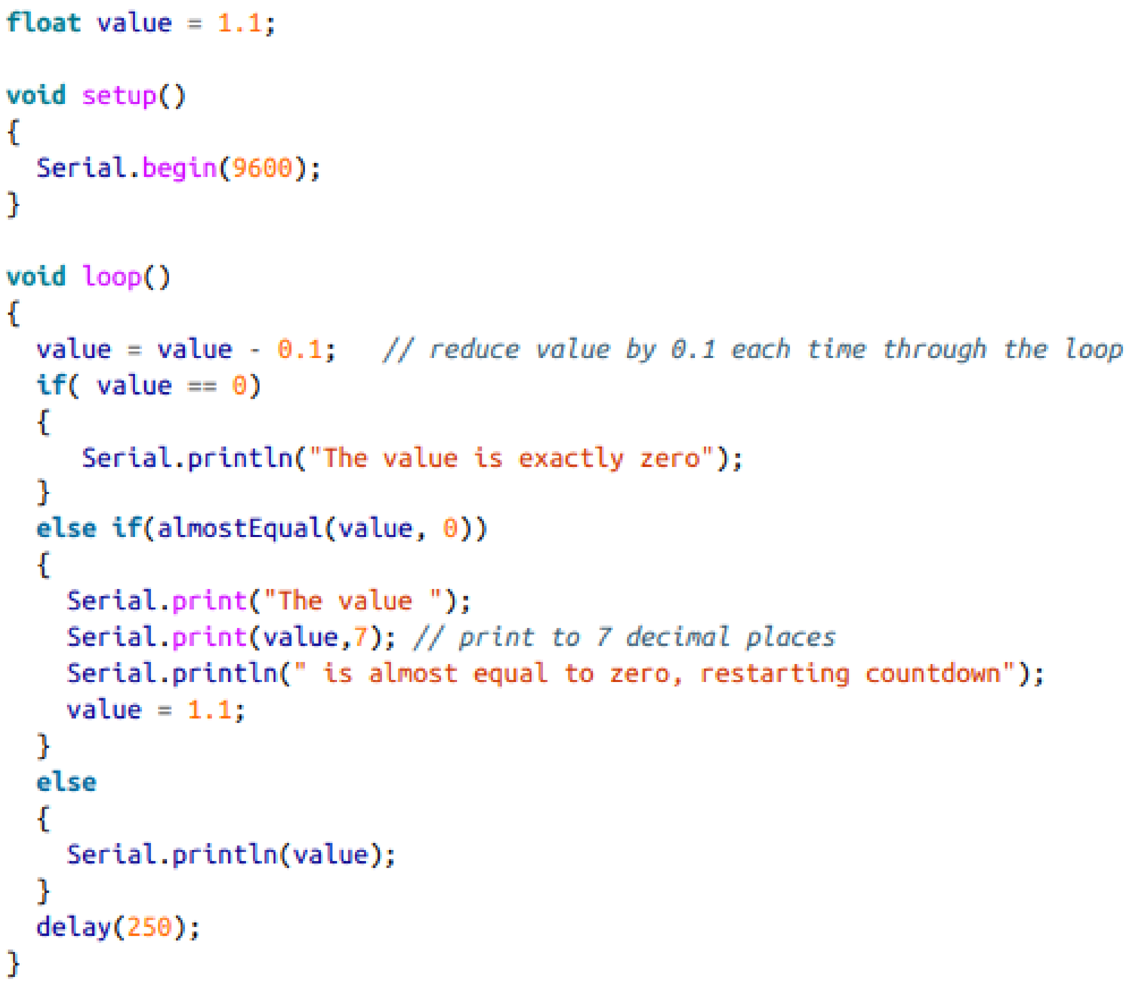
Other types	Use
`String`	Represents a sequence of characters typically used to contain text.
`void`	Used only in function declarations where no value is returned.

Using Simple Primitive Types (Variables)

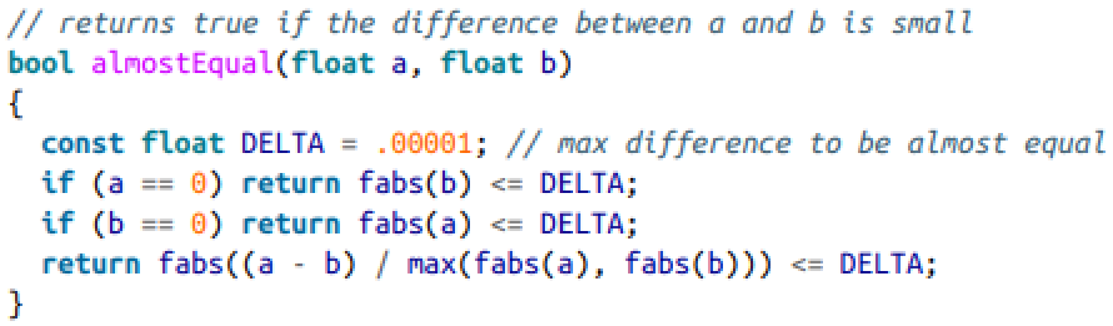
- Variables declared using int will be suitable for numeric values if the values do not exceed the range.
- Choose a type that specifically suits your application. This is especially important if you are calling library functions that return values other than int.
- bool (boolean) types have two possible values: true or false.

Using Floating-Point Numbers

- Floating-point numbers are used for values expressed with decimal points (this is the way to represent fractional values).

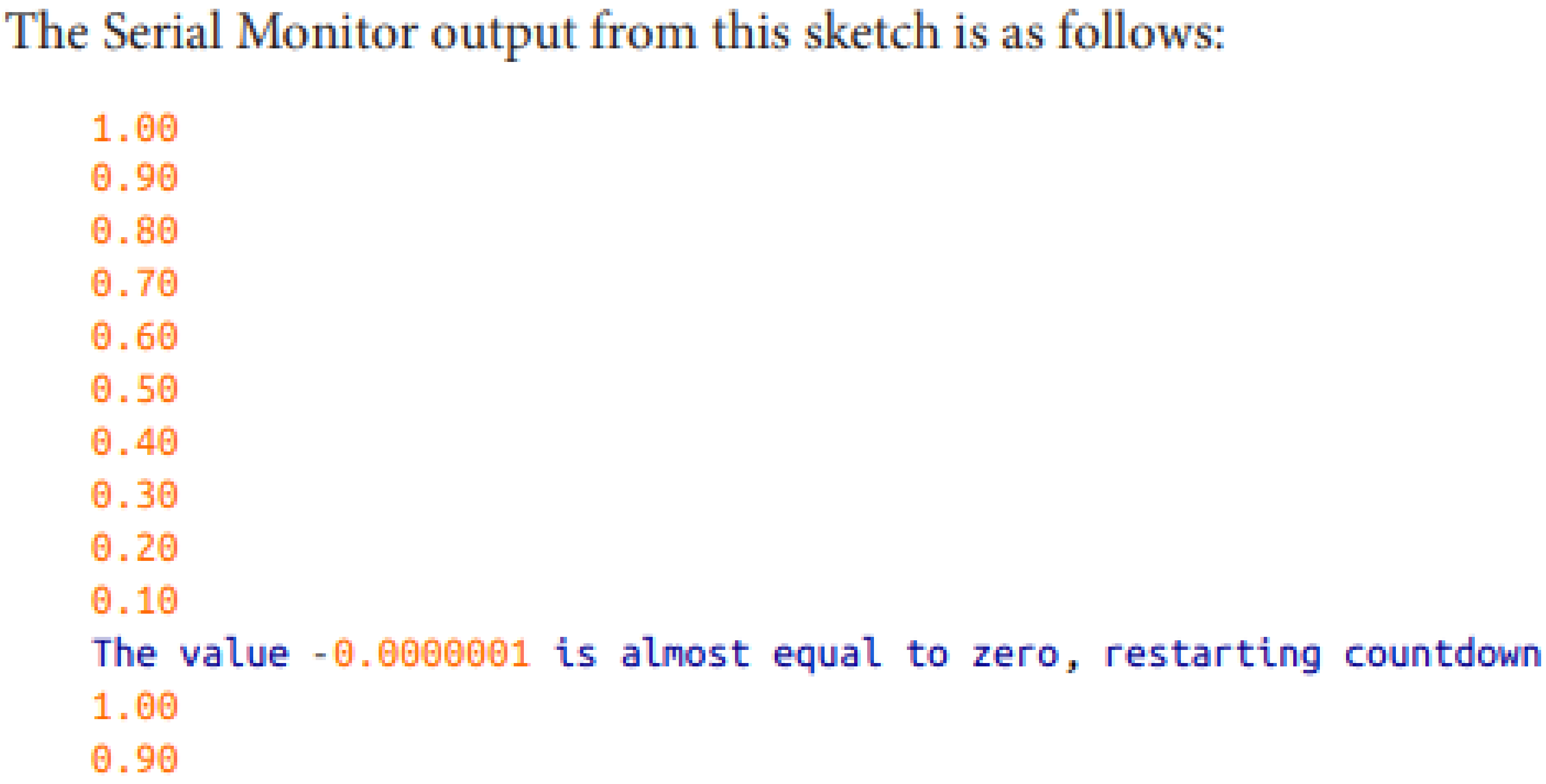


```
float value = 1.1;

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

void loop()
{
  value = value - 0.1;   // reduce value by 0.1 each time through the loop
  if( value == 0)
  {
     Serial.println("The value is exactly zero");
  }
  else if(almostEqual(value, 0))
  {
    Serial.print("The value ");
    Serial.print(value,7); // print to 7 decimal places
    Serial.println(" is almost equal to zero, restarting countdown");
    value = 1.1;
  }
  else
  {
    Serial.println(value);
  }
  delay(250);
}
```



```
// returns true if the difference between a and b is small
bool almostEqual(float a, float b)
{
  const float DELTA = .00001; // max difference to be almost equal
  if (a == 0) return fabs(b) <= DELTA;
  if (b == 0) return fabs(a) <= DELTA;
  return fabs((a - b) / max(fabs(a), fabs(b))) <= DELTA;
}
```

Using Floating-Point Numbers



The Serial Monitor output from this sketch is as follows:

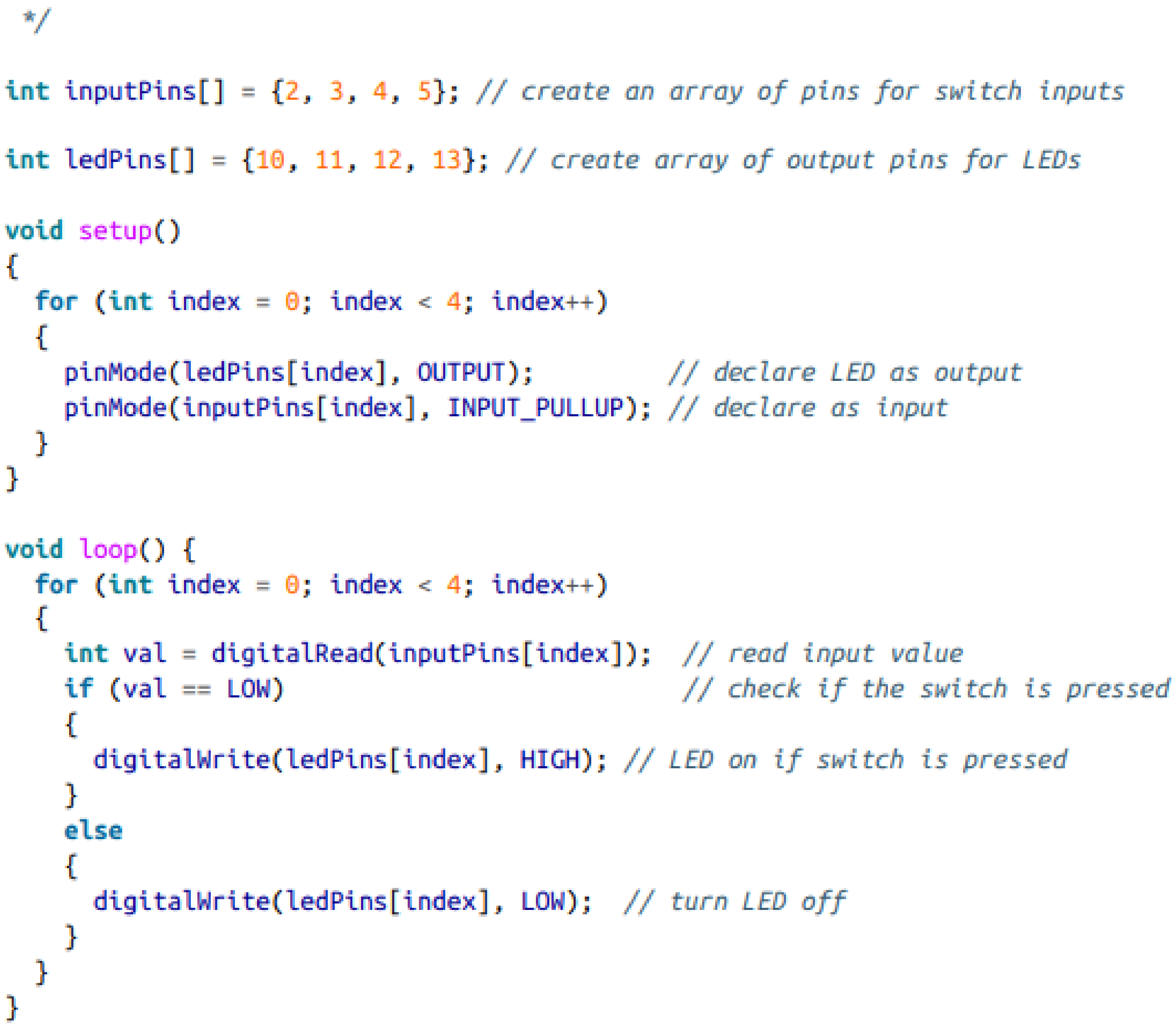
```
1.00
0.90
0.80
0.70
0.60
0.50
0.40
0.30
0.20
0.10
The value -0.0000001 is almost equal to zero, restarting countdown
1.00
0.90
```

Using Floating-Point Numbers

- You may expect the code to print "The value is exactly zero" after value is 0.1 and then 0.1 is subtracted from it.
- But value never equals exactly zero; it gets very close, but that is not good enough to pass the test: if (value == 0).
- This is because the only memory-efficient way that floating-point numbers can contain the huge range in values they can represent is by storing an approximation of the number.

Working with Groups of Values

- You want to create and use a group of values (called arrays). The arrays may be a simple list or they could have two or more dimensions.

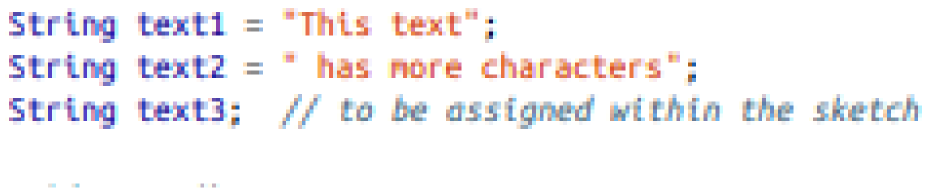


```
 */

int inputPins[] = {2, 3, 4, 5}; // create an array of pins for switch inputs

int ledPins[] = {10, 11, 12, 13}; // create array of output pins for LEDs

void setup()
{
  for (int index = 0; index < 4; index++)
  {
    pinMode(ledPins[index], OUTPUT);         // declare LED as output
    pinMode(inputPins[index], INPUT_PULLUP); // declare as input
  }
}

void loop() {
  for (int index = 0; index < 4; index++)
  {
    int val = digitalRead(inputPins[index]);  // read input value
    if (val == LOW)                           // check if the switch is pressed
    {
      digitalWrite(ledPins[index], HIGH); // LED on if switch is pressed
    }
    else
    {
      digitalWrite(ledPins[index], LOW);  // turn LED off
    }
  }
}
```

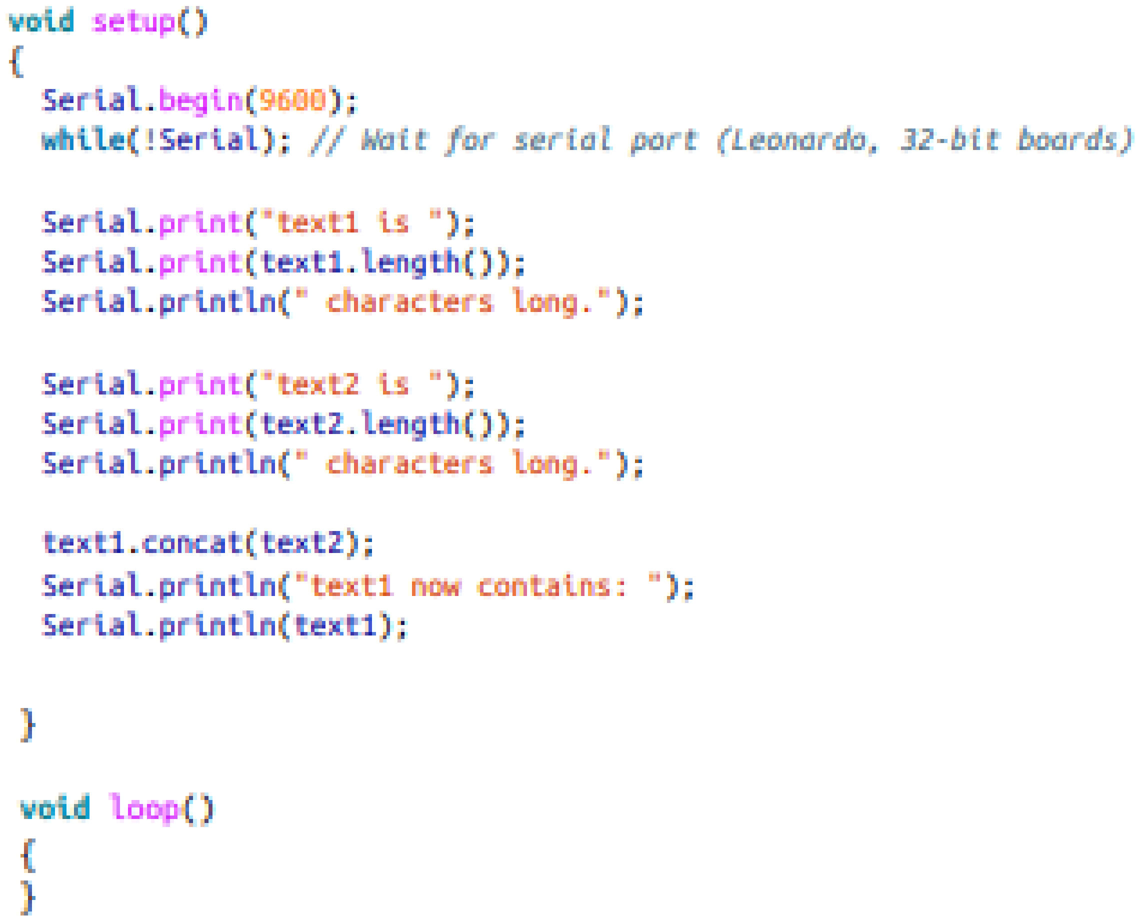
- INPUT_PULLUP mode enables Arduino’s internal pull-up resistors. The difference with the INPUT_PULLUP mode is that when the button is pressed, digital Read returns LOW rather than HIGH.

Using Arduino String Functionality

- You want to manipulate text. You need to copy it, add bits together, and determine the number of characters.



```
String text1 = "This text";
String text2 = " has more characters";
String text3;  // to be assigned within the sketch
```



```
void setup()
{
  Serial.begin(9600);
  while(!Serial); // Wait for serial port (Leonardo, 32-bit boards)

  Serial.print("text1 is ");
  Serial.print(text1.length());
  Serial.println(" characters long.");

  Serial.print("text2 is ");
  Serial.print(text2.length());
  Serial.println(" characters long.");

  text1.concat(text2);
  Serial.println("text1 now contains: ");
  Serial.println(text1);

}

void loop()
{
}
```

Using Arduino String Functionality

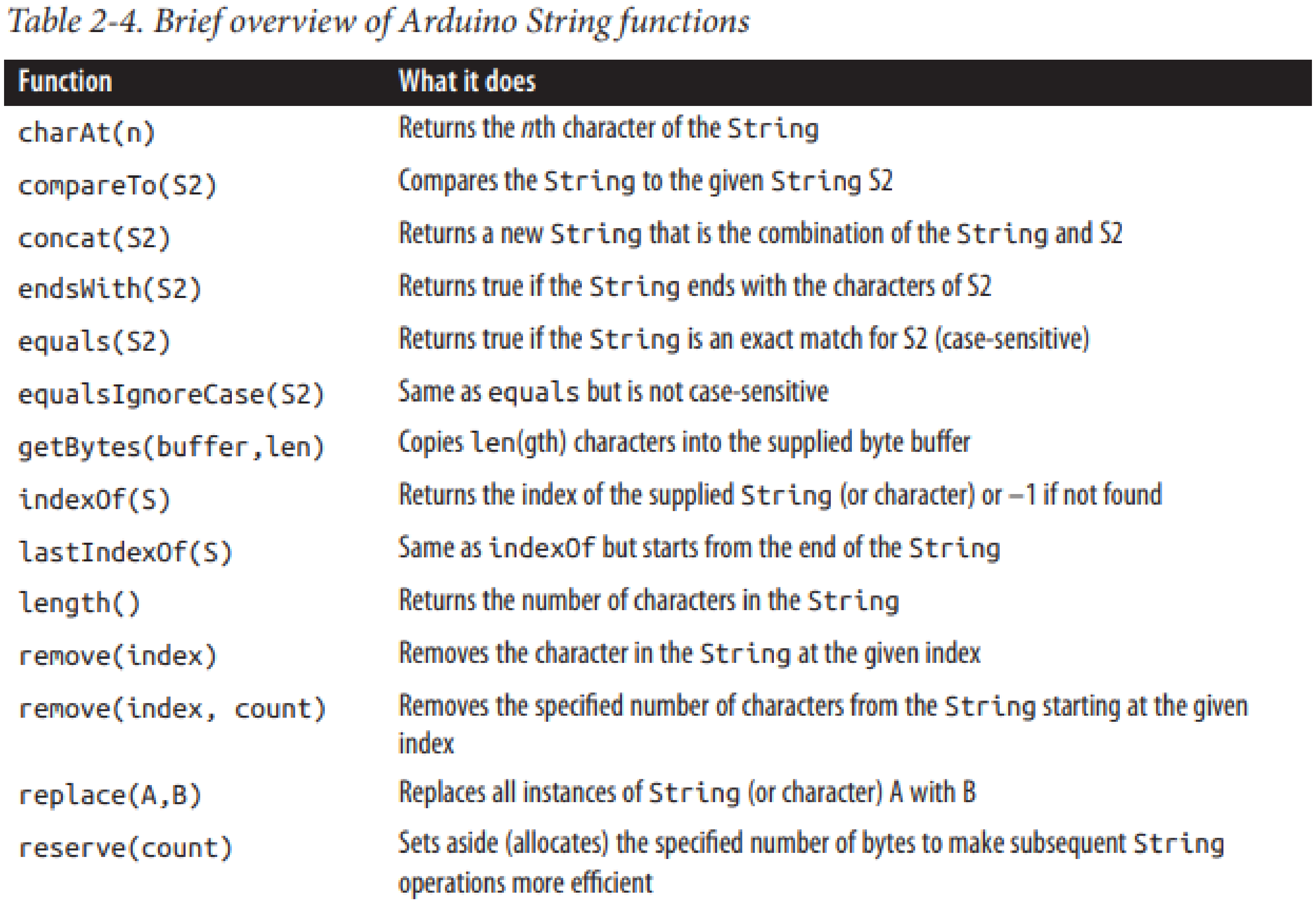
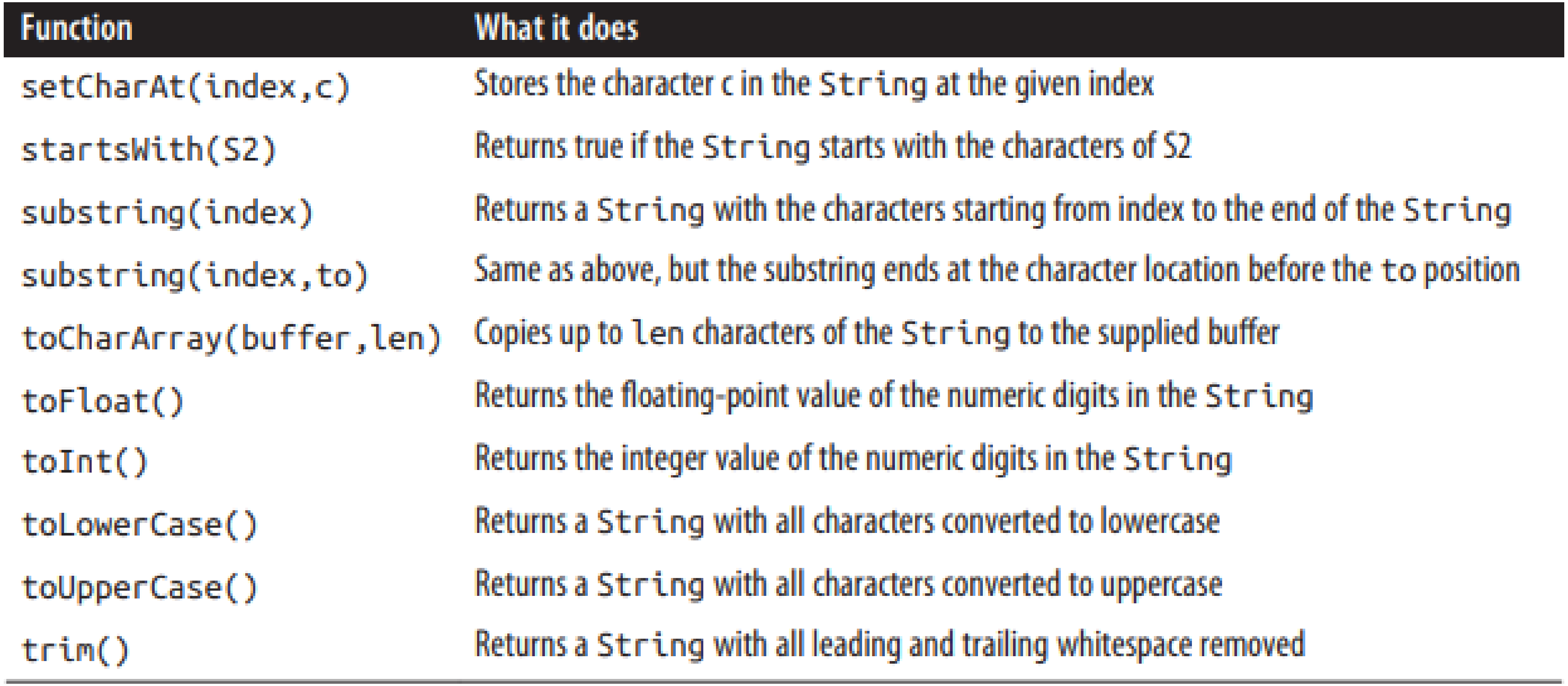


Table 2-4. Brief overview of Arduino String functions

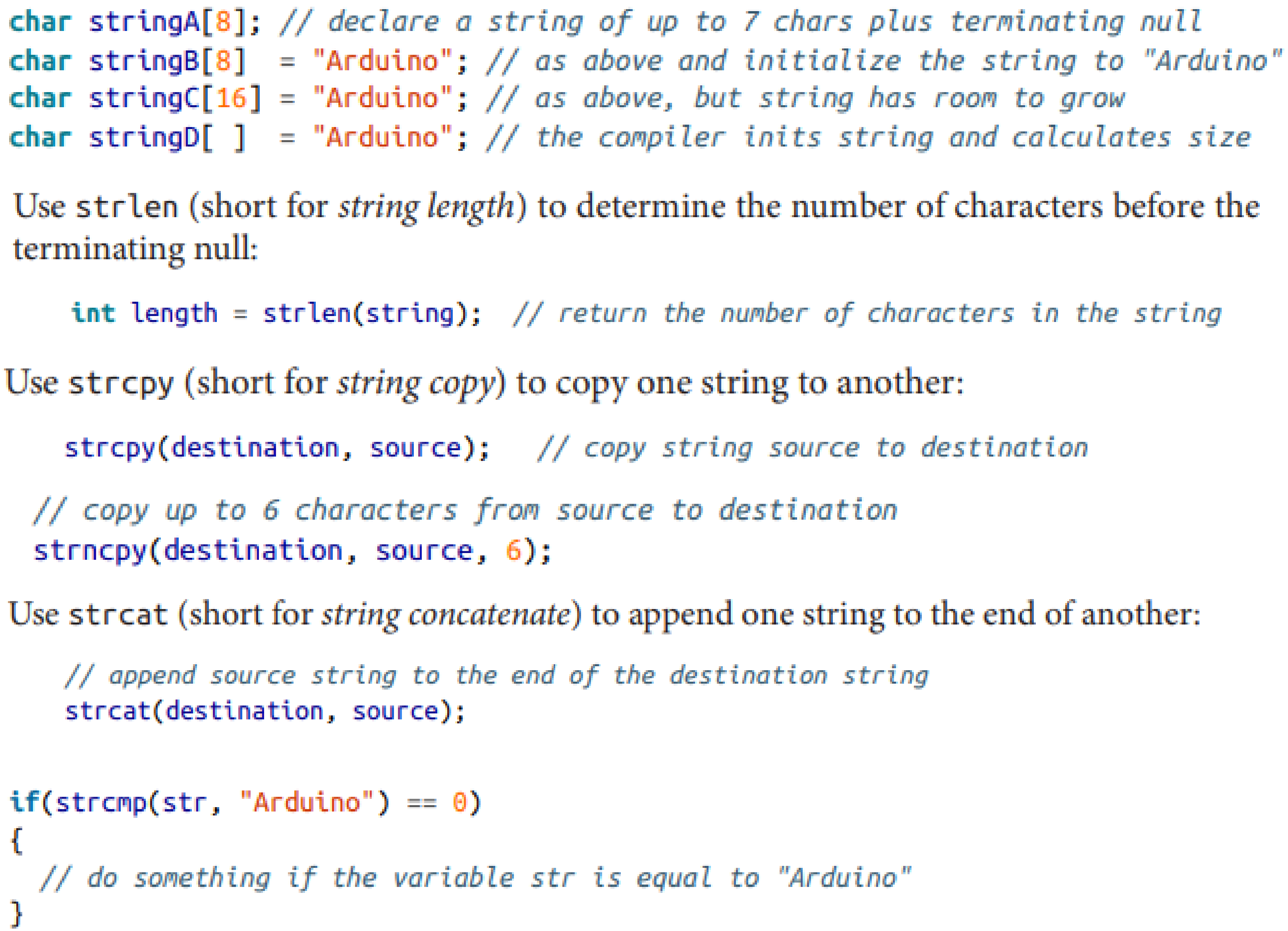
Function	What it does
`charAt(n)`	Returns the *n*th character of the `String`
`compareTo(S2)`	Compares the `String` to the given `String` S2
`concat(S2)`	Returns a new `String` that is the combination of the `String` and S2
`endsWith(S2)`	Returns true if the `String` ends with the characters of S2
`equals(S2)`	Returns true if the `String` is an exact match for S2 (case-sensitive)
`equalsIgnoreCase(S2)`	Same as `equals` but is not case-sensitive
`getBytes(buffer,len)`	Copies `len`(gth) characters into the supplied byte buffer
`indexOf(S)`	Returns the index of the supplied `String` (or character) or –1 if not found
`lastIndexOf(S)`	Same as `indexOf` but starts from the end of the `String`
`length()`	Returns the number of characters in the `String`
`remove(index)`	Removes the character in the `String` at the given index
`remove(index, count)`	Removes the specified number of characters from the `String` starting at the given index
`replace(A,B)`	Replaces all instances of `String` (or character) A with B
`reserve(count)`	Sets aside (allocates) the specified number of bytes to make subsequent `String` operations more efficient

Using Arduino String Functionality



Function	What it does
`setCharAt(index,c)`	Stores the character c in the `String` at the given index
`startsWith(S2)`	Returns true if the `String` starts with the characters of S2
`substring(index)`	Returns a `String` with the characters starting from index to the end of the `String`
`substring(index,to)`	Same as above, but the substring ends at the character location before the `to` position
`toCharArray(buffer,len)`	Copies up to `len` characters of the `String` to the supplied buffer
`toFloat()`	Returns the floating-point value of the numeric digits in the `String`
`toInt()`	Returns the integer value of the numeric digits in the `String`
`toLowerCase()`	Returns a `String` with all characters converted to lowercase
`toUpperCase()`	Returns a `String` with all characters converted to uppercase
`trim()`	Returns a `String` with all leading and trailing whitespace removed

Using C Character Strings



```
char stringA[8]; // declare a string of up to 7 chars plus terminating null
char stringB[8]  = "Arduino"; // as above and initialize the string to "Arduino"
char stringC[16] = "Arduino"; // as above, but string has room to grow
char stringD[ ]  = "Arduino"; // the compiler inits string and calculates size
```

Use `strlen` (short for *string length*) to determine the number of characters before the terminating null:

```
int length = strlen(string);  // return the number of characters in the string
```

Use `strcpy` (short for *string copy*) to copy one string to another:

```
strcpy(destination, source);   // copy string source to destination
```

```
// copy up to 6 characters from source to destination
strncpy(destination, source, 6);
```

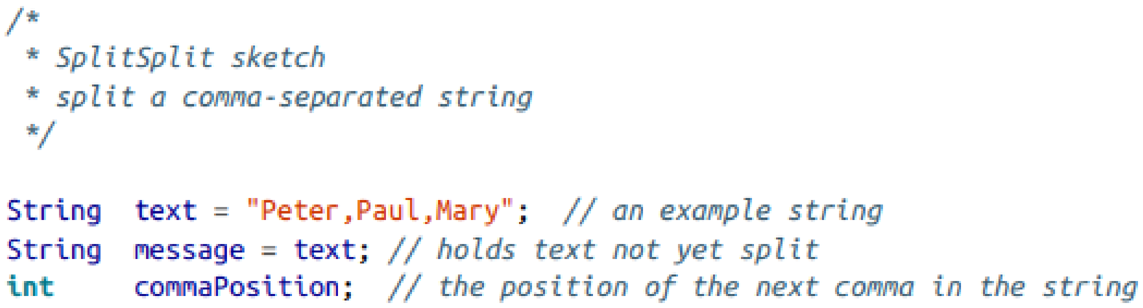
Use `strcat` (short for *string concatenate*) to append one string to the end of another:

```
// append source string to the end of the destination string
strcat(destination, source);
```

```
if(strcmp(str, "Arduino") == 0)
{
  // do something if the variable str is equal to "Arduino"
}
```

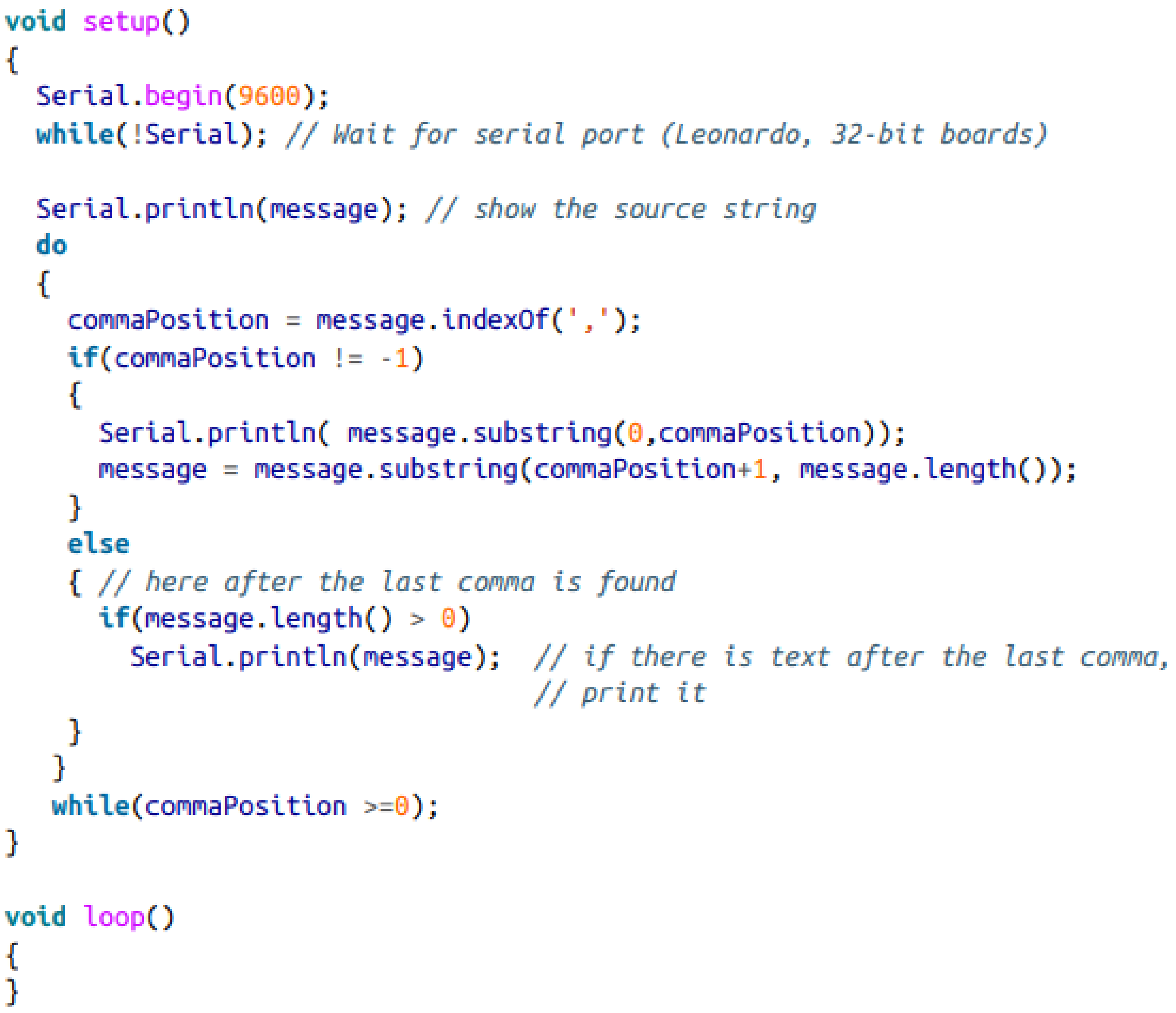
Splitting Comma-Separated Text into Groups

- You have a string that contains two or more pieces of data separated by commas (or any other separator). You want to split the string so that you can use each individual part.



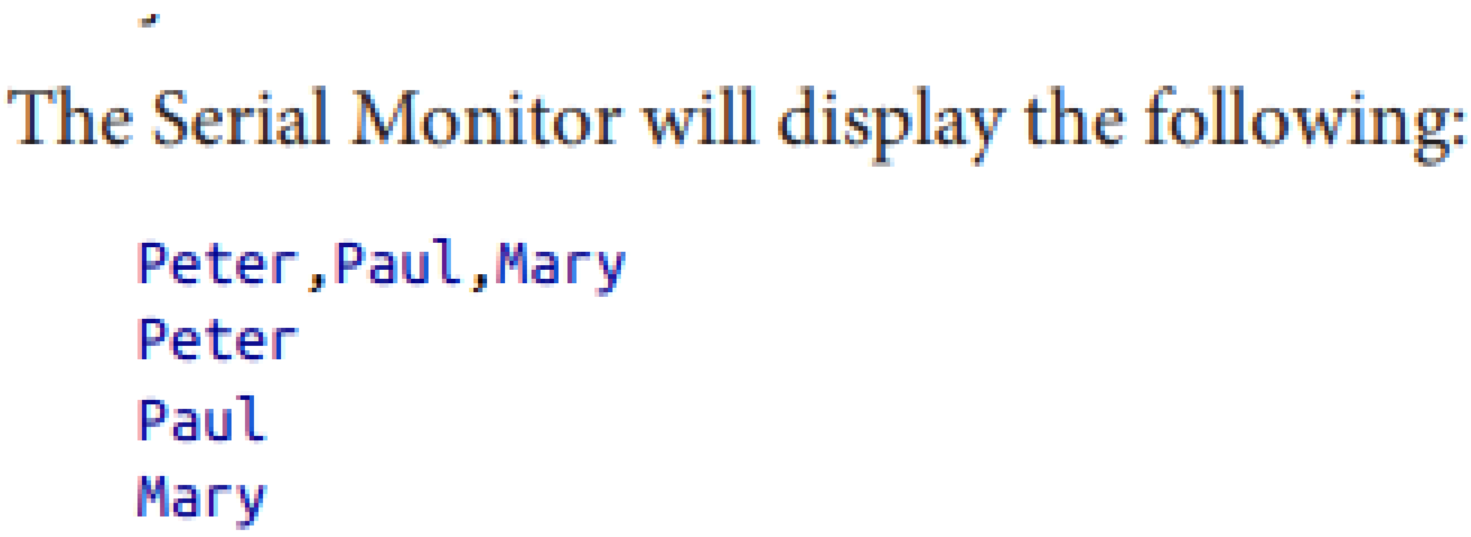
```
/*
 * SplitSplit sketch
 * split a comma-separated string
 */

String  text = "Peter,Paul,Mary";  // an example string
String  message = text; // holds text not yet split
int     commaPosition;  // the position of the next comma in the string
```

Splitting Comma-Separated Text into Groups



```
void setup()
{
  Serial.begin(9600);
  while(!Serial); // Wait for serial port (Leonardo, 32-bit boards)

  Serial.println(message); // show the source string
  do
  {
    commaPosition = message.indexOf(',');
    if(commaPosition != -1)
    {
      Serial.println( message.substring(0,commaPosition));
      message = message.substring(commaPosition+1, message.length());
    }
    else
    { // here after the last comma is found
      if(message.length() > 0)
        Serial.println(message);  // if there is text after the last comma,
                                  // print it
    }
   }
   while(commaPosition >=0);
}

void loop()
{
}
```

Splitting Comma-Separated Text into Groups

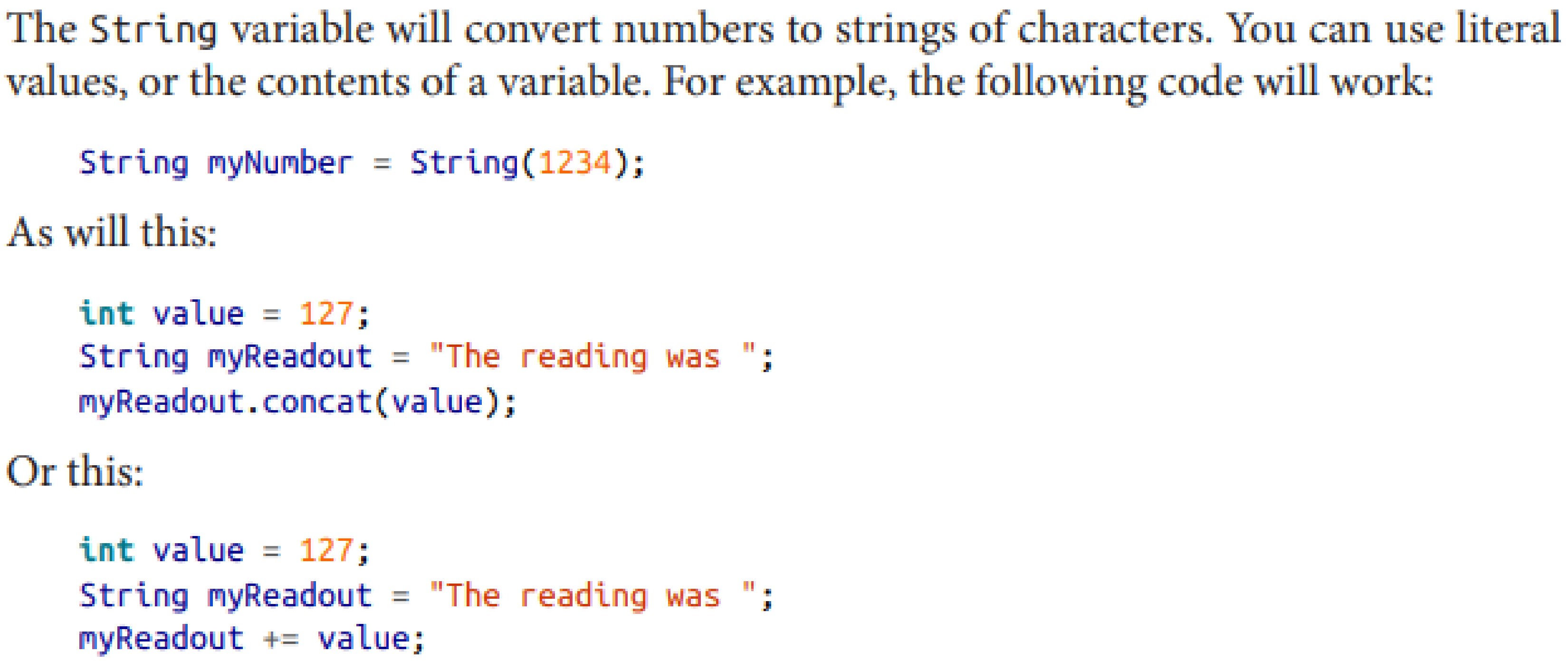


The Serial Monitor will display the following:

```
Peter,Paul,Mary
Peter
Paul
Mary
```

Converting a Number to a String

- You need to convert a number to a string, perhaps to show the number on an LCD or other display



The `String` variable will convert numbers to strings of characters. You can use literal values, or the contents of a variable. For example, the following code will work:

```
String myNumber = String(1234);
```

As will this:

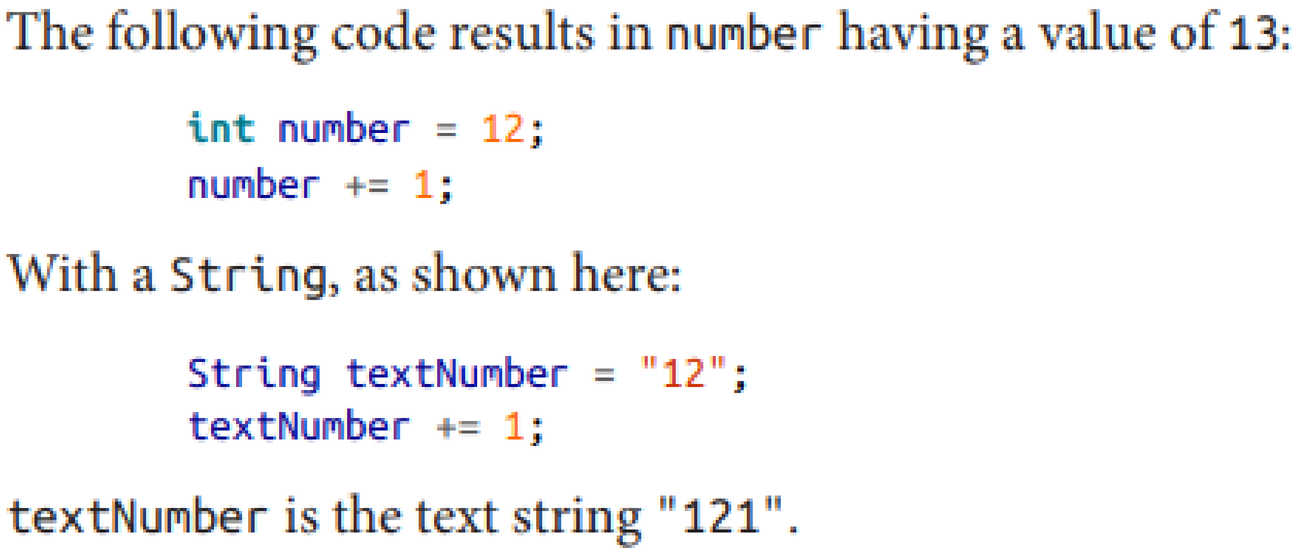
```
int value = 127;
String myReadout = "The reading was ";
myReadout.concat(value);
```

Or this:

```
int value = 127;
String myReadout = "The reading was ";
myReadout += value;
```

Converting a Number to a String

- The Arduino String class automatically converts numerical values when they are assigned to a String variable. You can combine (concatenate) numeric values at the end of a string using the concat function or the string + operator.



The following code results in number having a value of 13:

```
int number = 12;
number += 1;
```

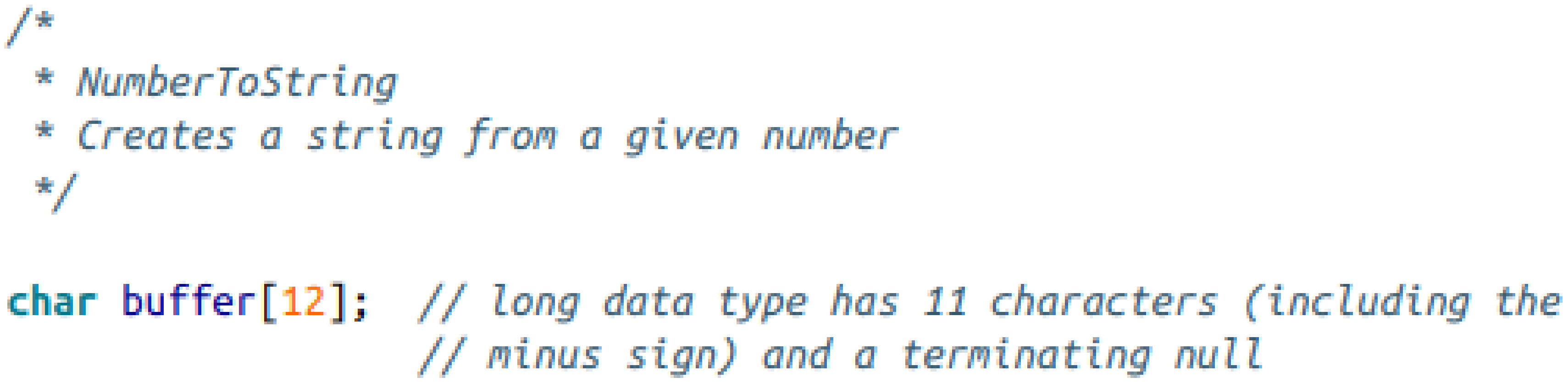
With a String, as shown here:

```
String textNumber = "12";
textNumber += 1;
```

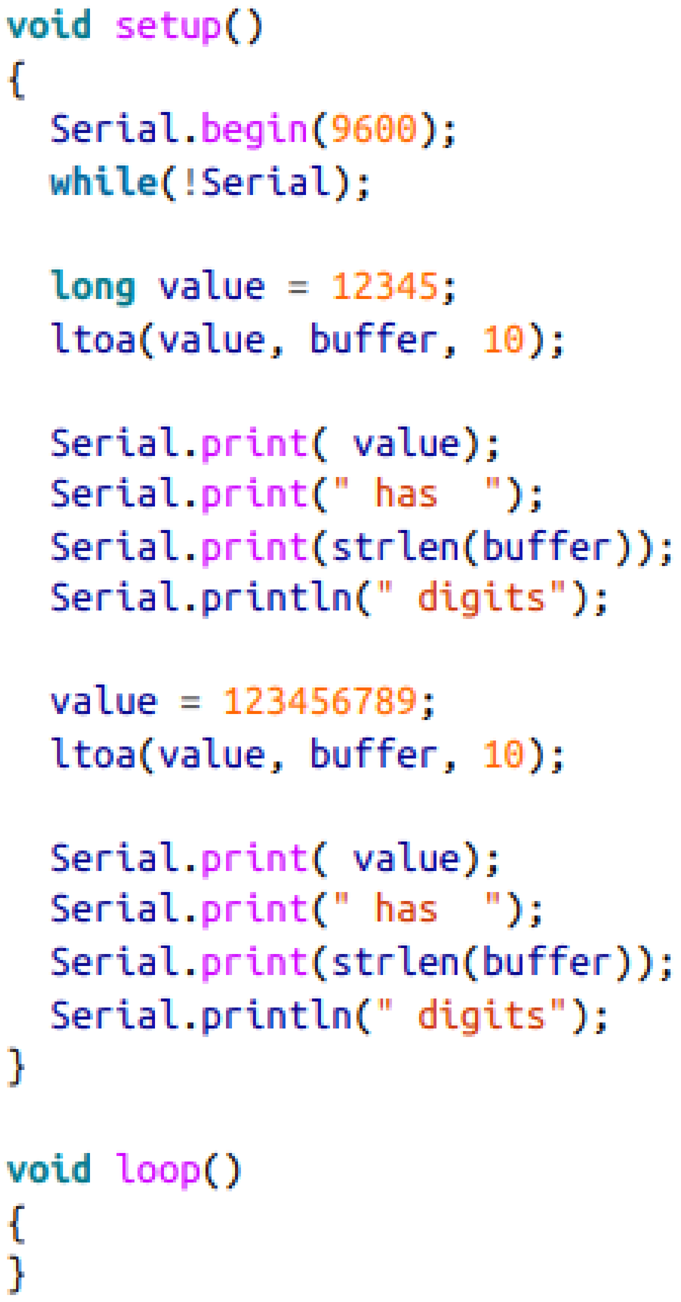
textNumber is the text string "121".

Itoa and ltoa

- itoa and ltoa take three parameters: the value to convert, the buffer that will hold the output string, and the number base (10 for a decimal number, 16 for hex, and 2 for binary).



```
/*
 * NumberToString
 * Creates a string from a given number
 */

char buffer[12];  // long data type has 11 characters (including the
                  // minus sign) and a terminating null
```

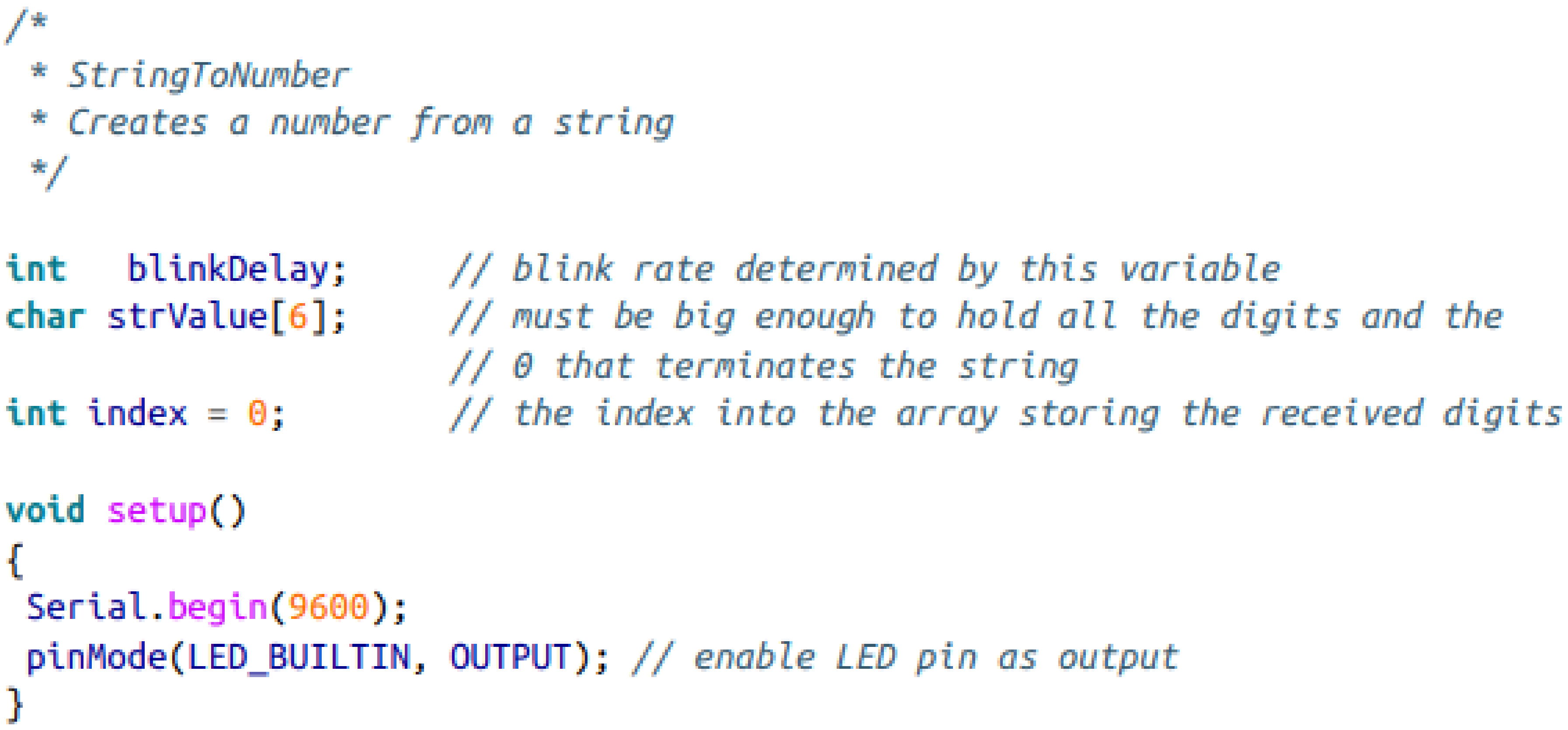


```
void setup()
{
  Serial.begin(9600);
  while(!Serial);

  long value = 12345;
  ltoa(value, buffer, 10);

  Serial.print( value);
  Serial.print(" has  ");
  Serial.print(strlen(buffer));
  Serial.println(" digits");

  value = 123456789;
  ltoa(value, buffer, 10);

  Serial.print( value);
  Serial.print(" has  ");
  Serial.print(strlen(buffer));
  Serial.println(" digits");
}

void loop()
{
}
```

Converting a String to a Number

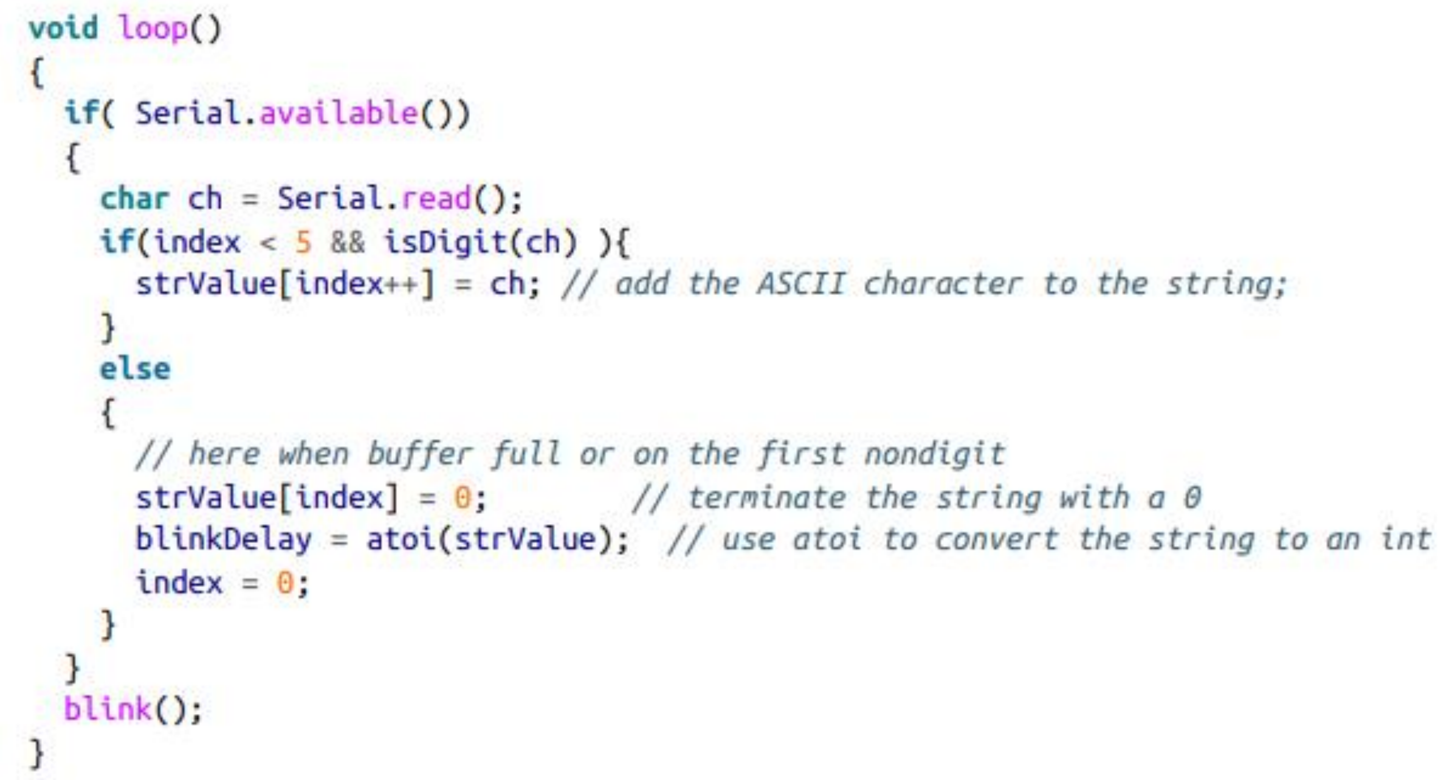
- You need to convert a string to a number. Perhaps you have received a value as a string over a communication link and you need to use this as an integer or floating-point value.
- There are a number of ways to solve this. If the string is received as serial stream data, it can be converted using the parseInt function.
- Another approach to converting text strings representing numbers is to use the C language conversion function called atoi (for int variables) or atol (for long variables).

Converting a String to a Number



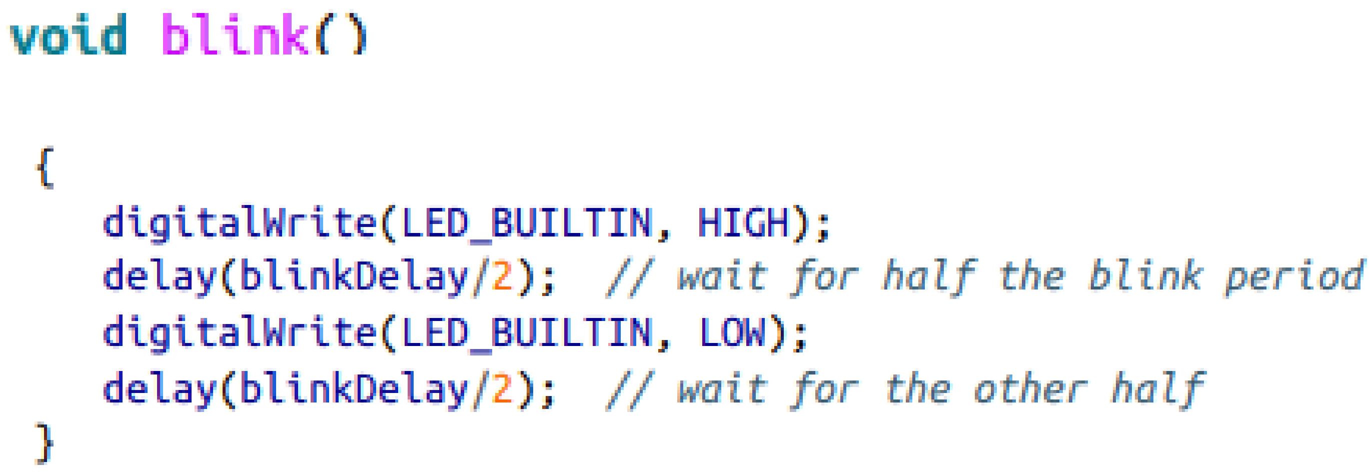
```
/*
 * StringToNumber
 * Creates a number from a string
 */

int   blinkDelay;     // blink rate determined by this variable
char strValue[6];     // must be big enough to hold all the digits and the
                      // 0 that terminates the string
int index = 0;        // the index into the array storing the received digits

void setup()
{
 Serial.begin(9600);
 pinMode(LED_BUILTIN, OUTPUT); // enable LED pin as output
}
```

Converting a String to a Number



```
void loop()
{
  if( Serial.available())
  {
    char ch = Serial.read();
    if(index < 5 && isDigit(ch) ){
      strValue[index++] = ch; // add the ASCII character to the string;
    }
    else
    {
      // here when buffer full or on the first nondigit
      strValue[index] = 0;        // terminate the string with a 0
      blinkDelay = atoi(strValue);  // use atoi to convert the string to an int
      index = 0;
    }
  }
  blink();
}
```

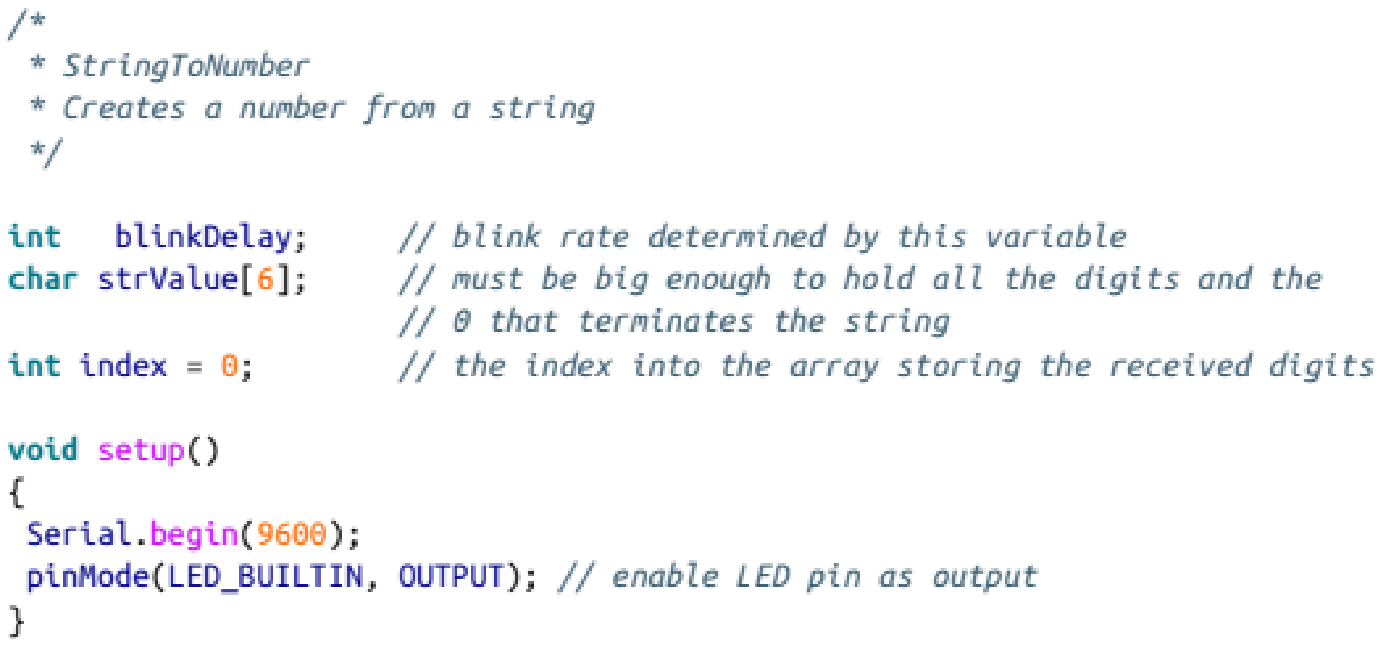
Converting a String to a Number



```
void blink()

 {
    digitalWrite(LED_BUILTIN, HIGH);
    delay(blinkDelay/2);  // wait for half the blink period
    digitalWrite(LED_BUILTIN, LOW);
    delay(blinkDelay/2);  // wait for the other half
 }
```

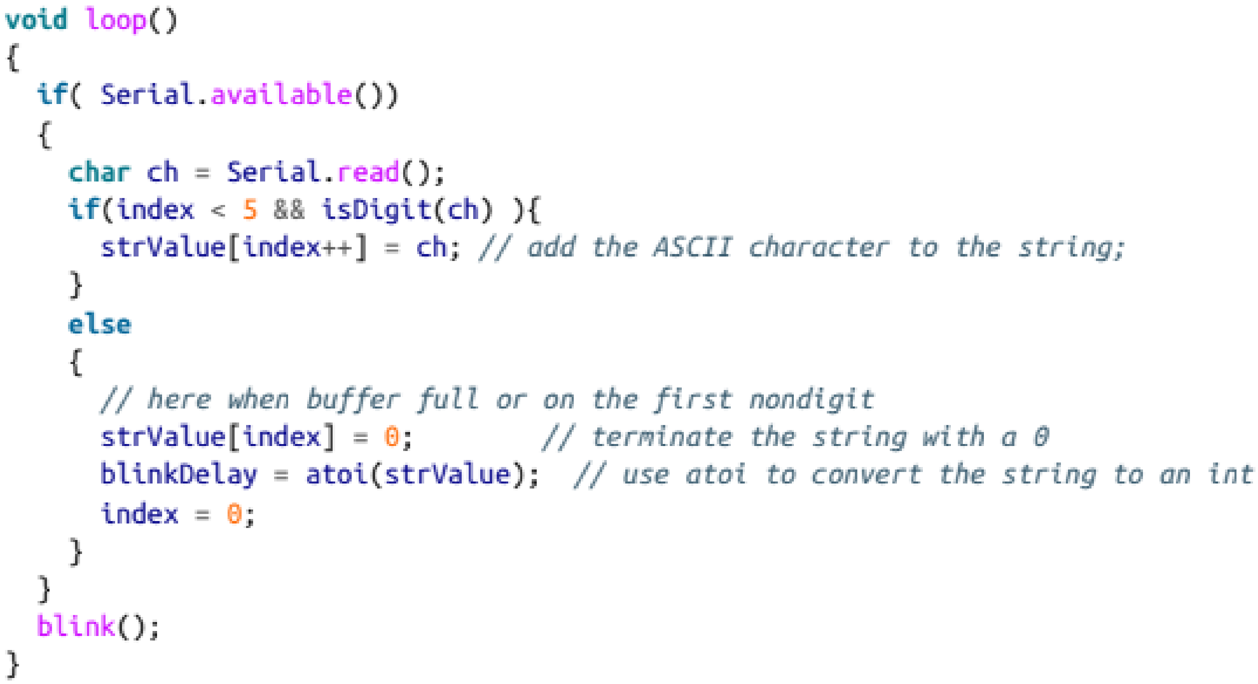
Converting a String to a Number

- You need to convert a string to a number. Perhaps you have received a value as a string over a communication link and you need to use this as an integer or floating point value.

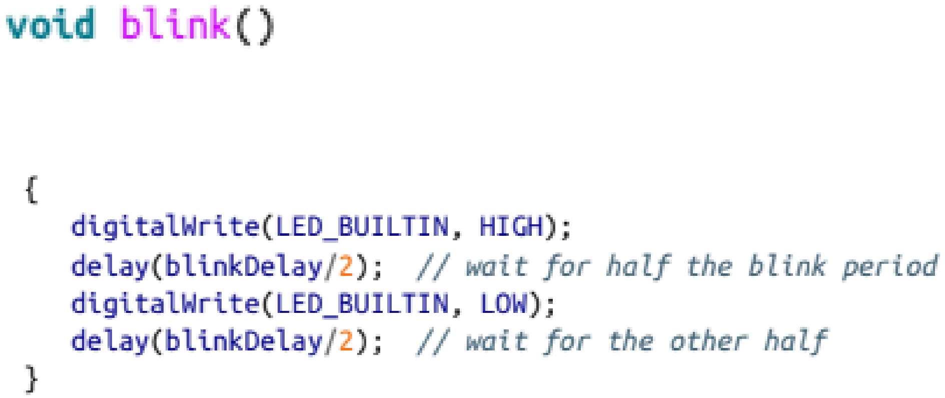


```
/*
 * StringToNumber
 * Creates a number from a string
 */

int   blinkDelay;     // blink rate determined by this variable
char strValue[6];     // must be big enough to hold all the digits and the
                      // 0 that terminates the string
int index = 0;        // the index into the array storing the received digits

void setup()
{
 Serial.begin(9600);
 pinMode(LED_BUILTIN, OUTPUT); // enable LED pin as output
}
```

Converting a String to a Number



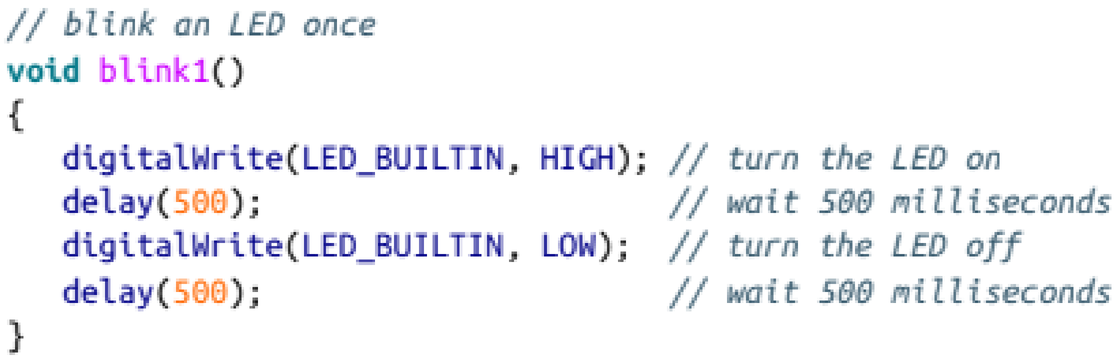
```
void loop()
{
  if( Serial.available())
  {
    char ch = Serial.read();
    if(index < 5 && isDigit(ch) ){
      strValue[index++] = ch; // add the ASCII character to the string;
    }
    else
    {
      // here when buffer full or on the first nondigit
      strValue[index] = 0;        // terminate the string with a 0
      blinkDelay = atoi(strValue);  // use atoi to convert the string to an int
      index = 0;
    }
  }
  blink();
}
```



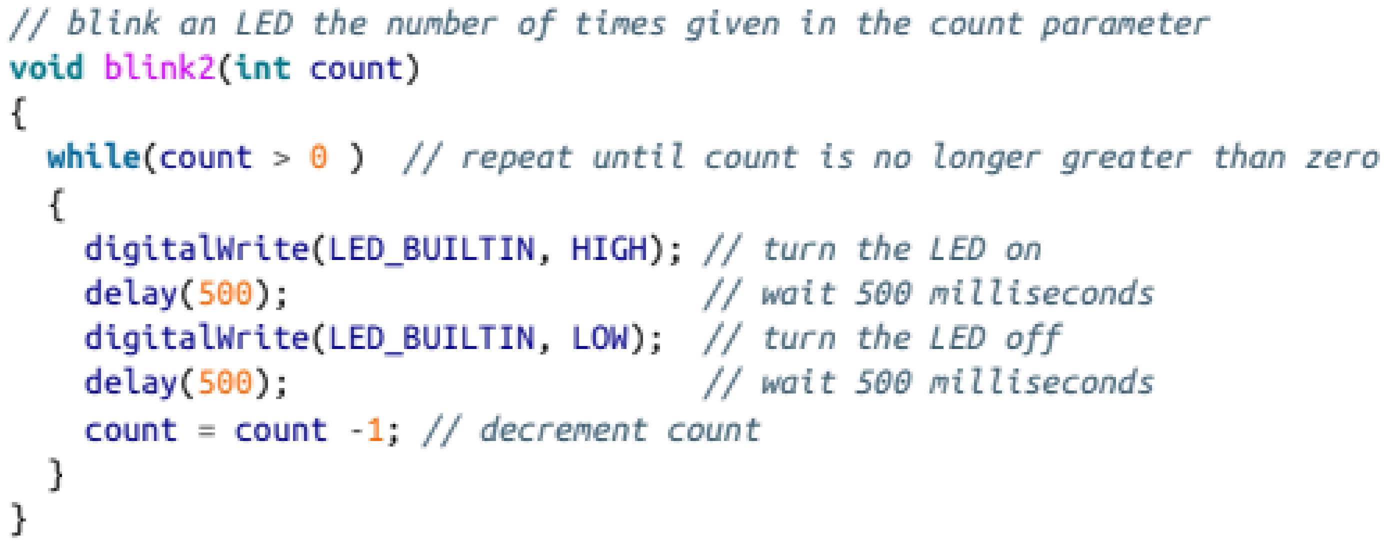
```
void blink()


{
   digitalWrite(LED_BUILTIN, HIGH);
   delay(blinkDelay/2);  // wait for half the blink period
   digitalWrite(LED_BUILTIN, LOW);
   delay(blinkDelay/2);  // wait for the other half
}
```

Structuring Your Code into Functional Blocks

- You want to know how to add functions to a sketch, and understand how to plan the overall structure of a sketch.



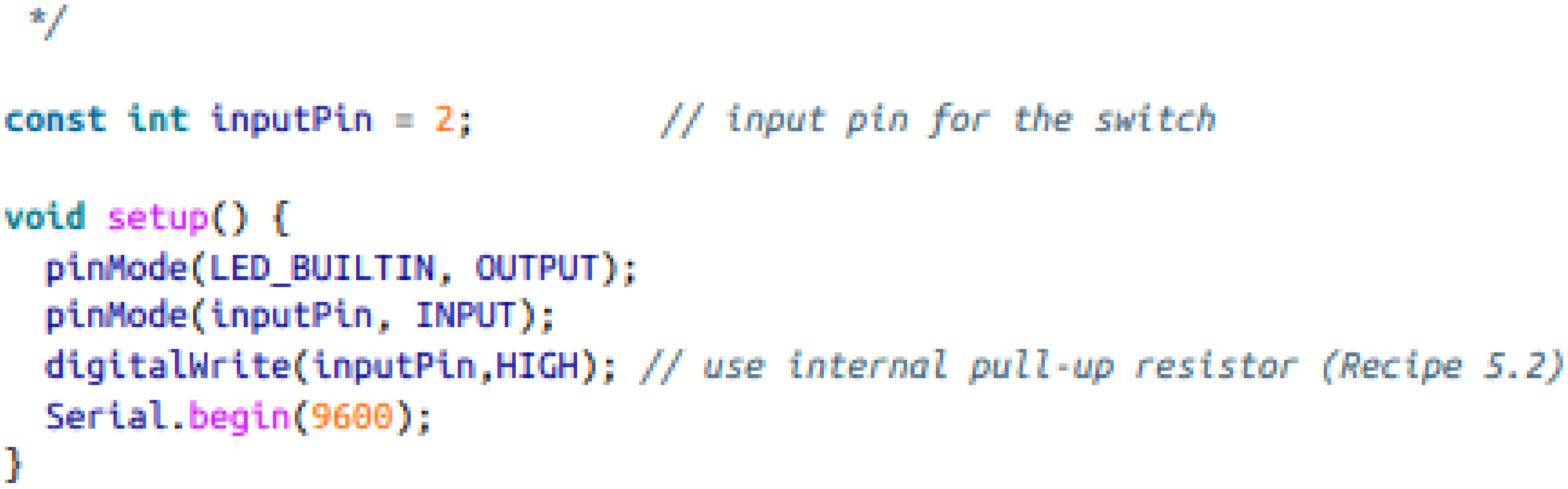
```
// blink an LED once
void blink1()
{
   digitalWrite(LED_BUILTIN, HIGH); // turn the LED on
   delay(500);                      // wait 500 milliseconds
   digitalWrite(LED_BUILTIN, LOW);  // turn the LED off
   delay(500);                      // wait 500 milliseconds
}
```



```
// blink an LED the number of times given in the count parameter
void blink2(int count)
{
  while(count > 0 )  // repeat until count is no longer greater than zero
  {
    digitalWrite(LED_BUILTIN, HIGH); // turn the LED on
    delay(500);                      // wait 500 milliseconds
    digitalWrite(LED_BUILTIN, LOW);  // turn the LED off
    delay(500);                      // wait 500 milliseconds
    count = count -1; // decrement count
  }
}
```

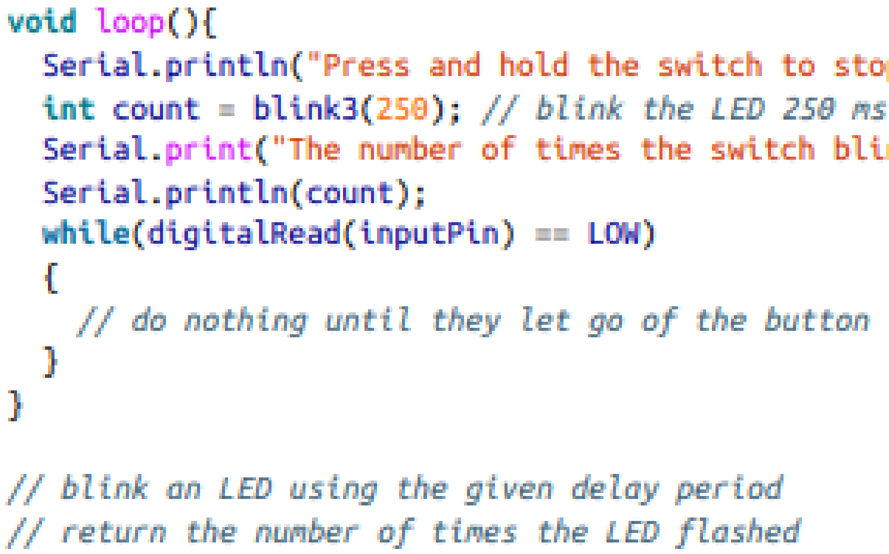
Example

- Write down a sketch with a function that takes a parameter and returns a value. The parameter determines the length of the LED on and off times (in milliseconds). The function continues to flash the LED until a button is pressed, and the number of times the LED flashed is returned from the function.

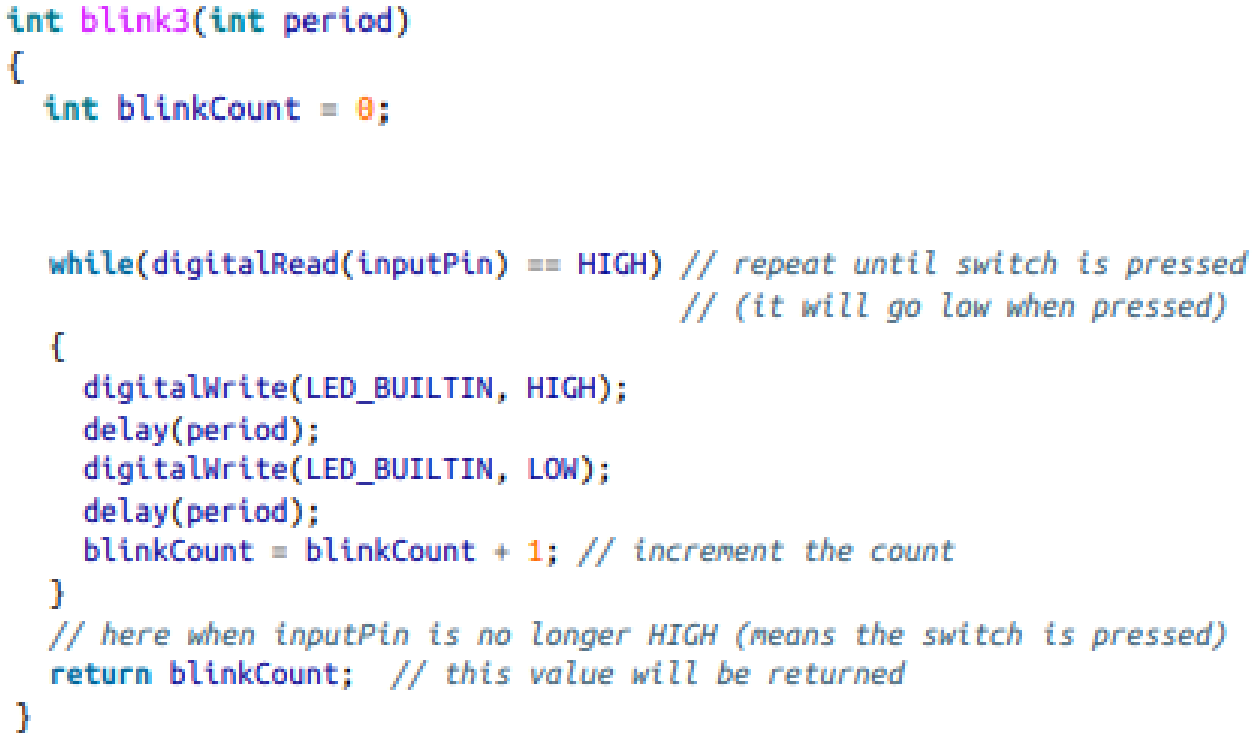


```
 */

const int inputPin = 2;          // input pin for the switch

void setup() {
  pinMode(LED_BUILTIN, OUTPUT);
  pinMode(inputPin, INPUT);
  digitalWrite(inputPin,HIGH); // use internal pull-up resistor (Recipe 5.2)
  Serial.begin(9600);
}
```

Example

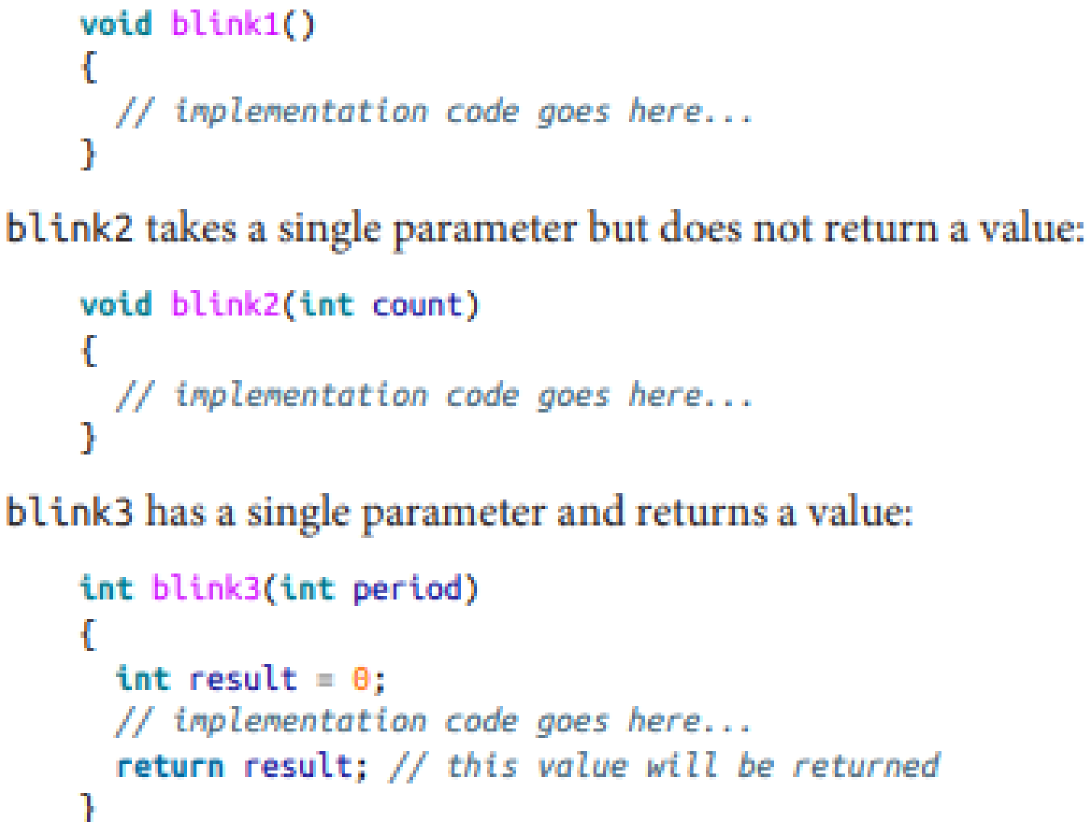


```
void loop(){
  Serial.println("Press and hold the switch to stop blinking");
  int count = blink3(250); // blink the LED 250 ms on and 250 ms off
  Serial.print("The number of times the switch blinked was ");
  Serial.println(count);
  while(digitalRead(inputPin) == LOW)
  {
    // do nothing until they let go of the button
  }
}

// blink an LED using the given delay period
// return the number of times the LED flashed
```



```
int blink3(int period)
{
  int blinkCount = 0;


  while(digitalRead(inputPin) == HIGH) // repeat until switch is pressed
                                       // (it will go low when pressed)
  {
    digitalWrite(LED_BUILTIN, HIGH);
    delay(period);
    digitalWrite(LED_BUILTIN, LOW);
    delay(period);
    blinkCount = blinkCount + 1; // increment the count
  }
  // here when inputPin is no longer HIGH (means the switch is pressed)
  return blinkCount;  // this value will be returned
}
```

Different Forms of Function



```
void blink1()
{
  // implementation code goes here...
}
```

blink2 takes a single parameter but does not return a value:

```
void blink2(int count)
{
  // implementation code goes here...
}
```

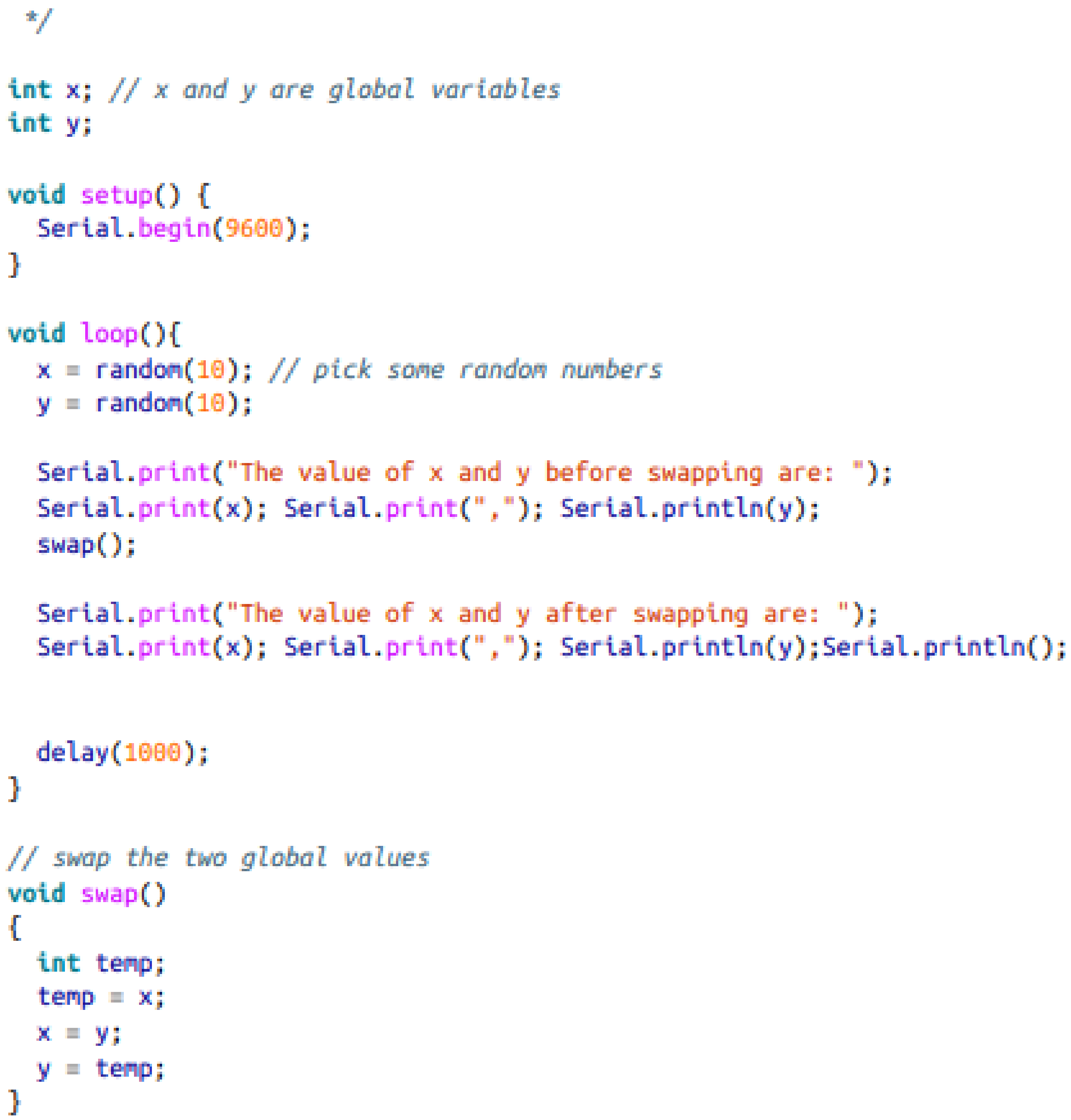
blink3 has a single parameter and returns a value:

```
int blink3(int period)
{
  int result = 0;
  // implementation code goes here...
  return result; // this value will be returned
}
```

Returning More than One Value from a Function

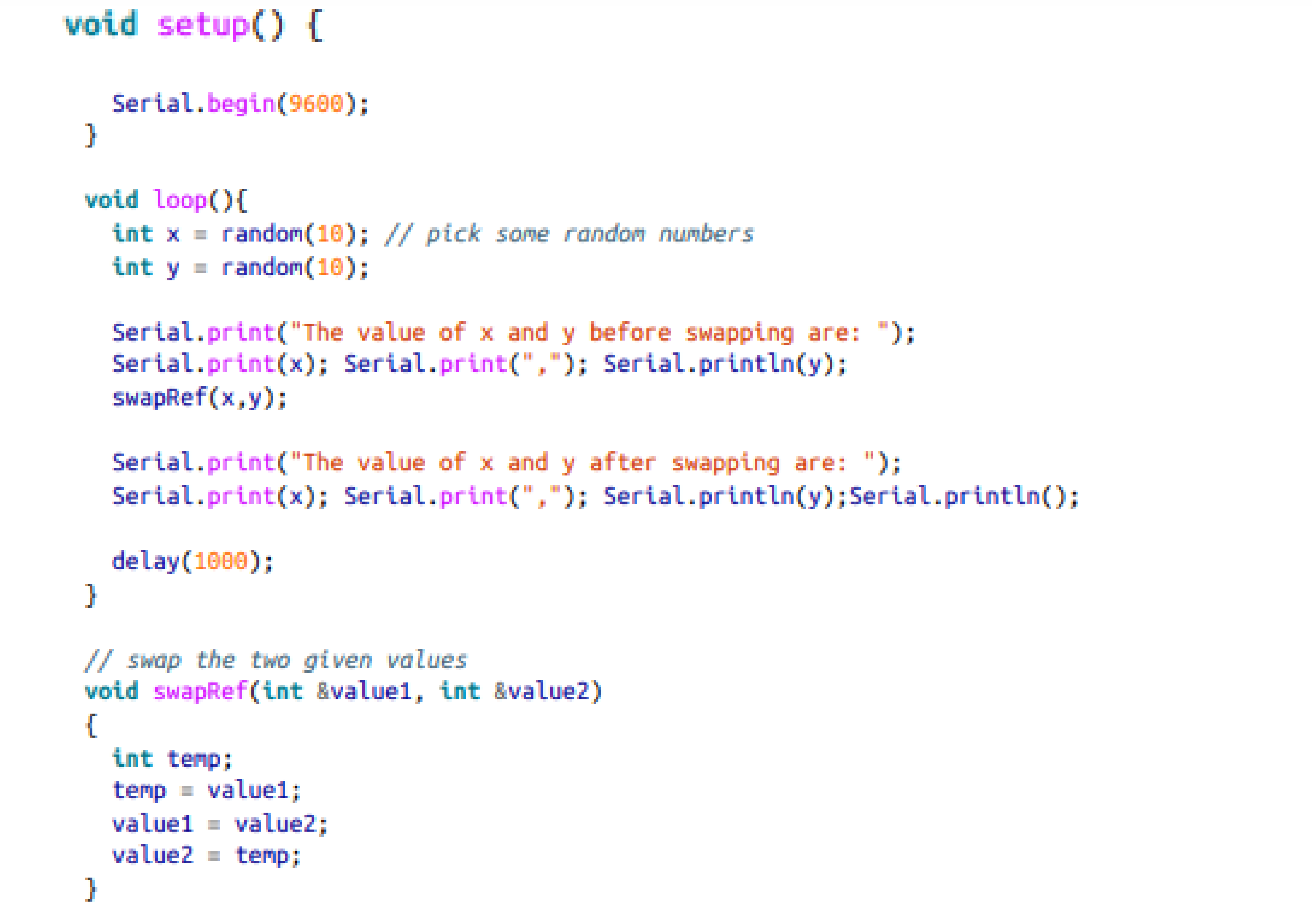
- You want to return two or more values from a function
- There are various ways to solve this. The easiest to understand is to have the function change some global variables and not actually return anything from the function.
- A safer and more elegant solution is to pass references to the values you want to change and let the function use the references to modify the values.

Returning More than One Value from a Function (Example 1)



```
 */

int x; // x and y are global variables
int y;

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

void loop(){
  x = random(10); // pick some random numbers
  y = random(10);

  Serial.print("The value of x and y before swapping are: ");
  Serial.print(x); Serial.print(","); Serial.println(y);
  swap();

  Serial.print("The value of x and y after swapping are: ");
  Serial.print(x); Serial.print(","); Serial.println(y);Serial.println();


  delay(1000);
}

// swap the two global values
void swap()
{
  int temp;
  temp = x;
  x = y;
  y = temp;
}
```

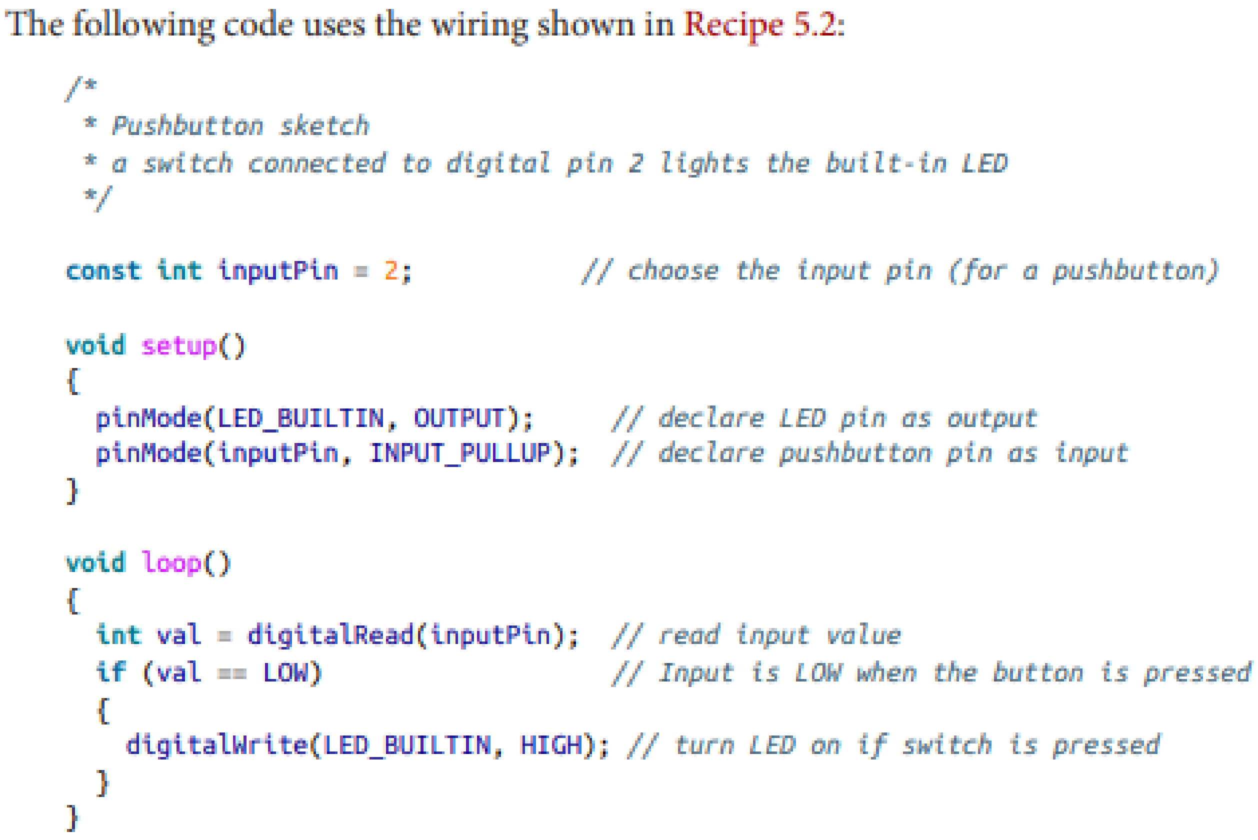
Returning More than One Value from a Function (Example 2)



```
  void setup() {

    Serial.begin(9600);
   }

   void loop(){
    int x = random(10); // pick some random numbers
    int y = random(10);

    Serial.print("The value of x and y before swapping are: ");
    Serial.print(x); Serial.print(","); Serial.println(y);
    swapRef(x,y);

    Serial.print("The value of x and y after swapping are: ");
    Serial.print(x); Serial.print(","); Serial.println(y);Serial.println();

    delay(1000);
   }

   // swap the two given values
   void swapRef(int &value1, int &value2)
   {
    int temp;
    temp = value1;
    value1 = value2;
    value2 = temp;
   }
```

Taking Actions Based on Conditions

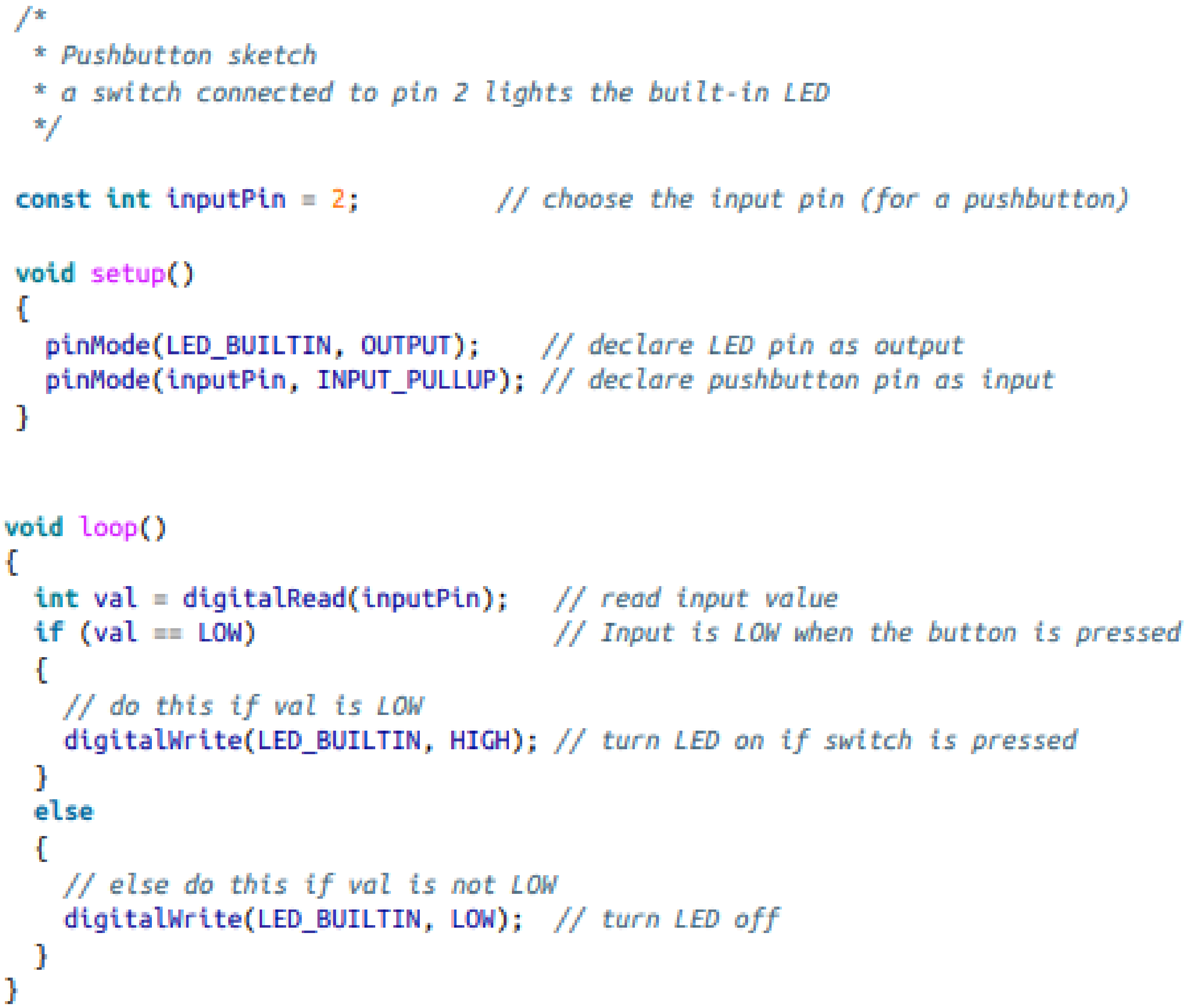
- You want to execute a block of code only if a particular condition is true. For example, you may want to light an LED if a switch is pressed or if an analog value is greater than some threshold.



The following code uses the wiring shown in Recipe 5.2:

```
/*
 * Pushbutton sketch
 * a switch connected to digital pin 2 lights the built-in LED
 */

const int inputPin = 2;           // choose the input pin (for a pushbutton)

void setup()
{
  pinMode(LED_BUILTIN, OUTPUT);     // declare LED pin as output
  pinMode(inputPin, INPUT_PULLUP);  // declare pushbutton pin as input
}

void loop()
{
  int val = digitalRead(inputPin);  // read input value
  if (val == LOW)                   // Input is LOW when the button is pressed
  {
    digitalWrite(LED_BUILTIN, HIGH); // turn LED on if switch is pressed
  }
}
```

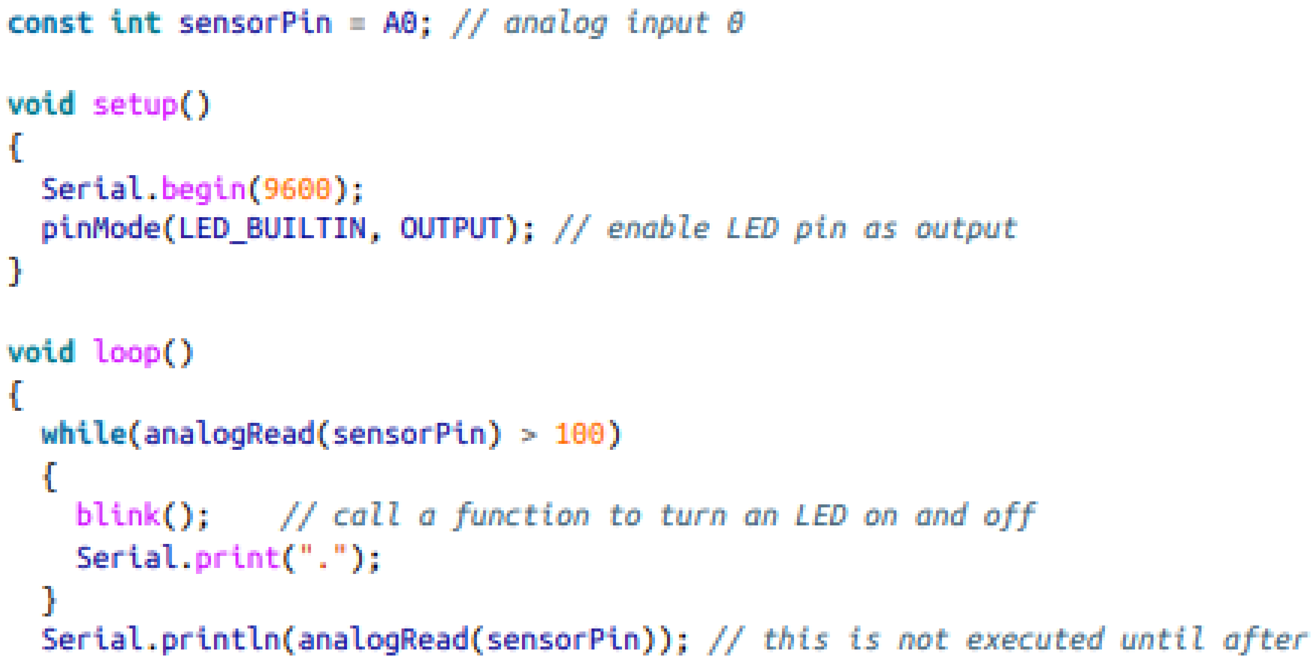
Taking Actions Based on Conditions



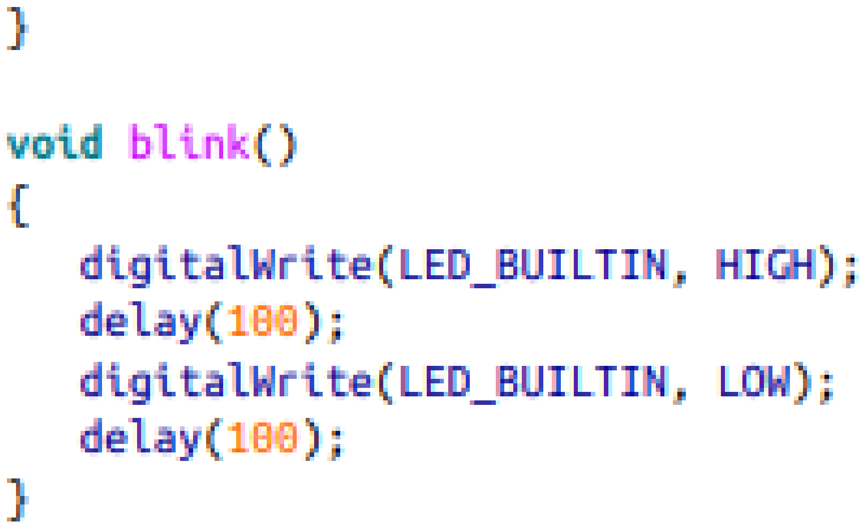
```
/*
 * Pushbutton sketch
 * a switch connected to pin 2 lights the built-in LED
 */

const int inputPin = 2;         // choose the input pin (for a pushbutton)

void setup()
{
  pinMode(LED_BUILTIN, OUTPUT);    // declare LED pin as output
  pinMode(inputPin, INPUT_PULLUP); // declare pushbutton pin as input
}


void loop()
{
  int val = digitalRead(inputPin);   // read input value
  if (val == LOW)                    // Input is LOW when the button is pressed
  {
    // do this if val is LOW
    digitalWrite(LED_BUILTIN, HIGH); // turn LED on if switch is pressed
  }
  else
  {
    // else do this if val is not LOW
    digitalWrite(LED_BUILTIN, LOW);  // turn LED off
  }
}
```

Repeating a Sequence of Statements

- You want to repeat a block of statements while an expression is true.



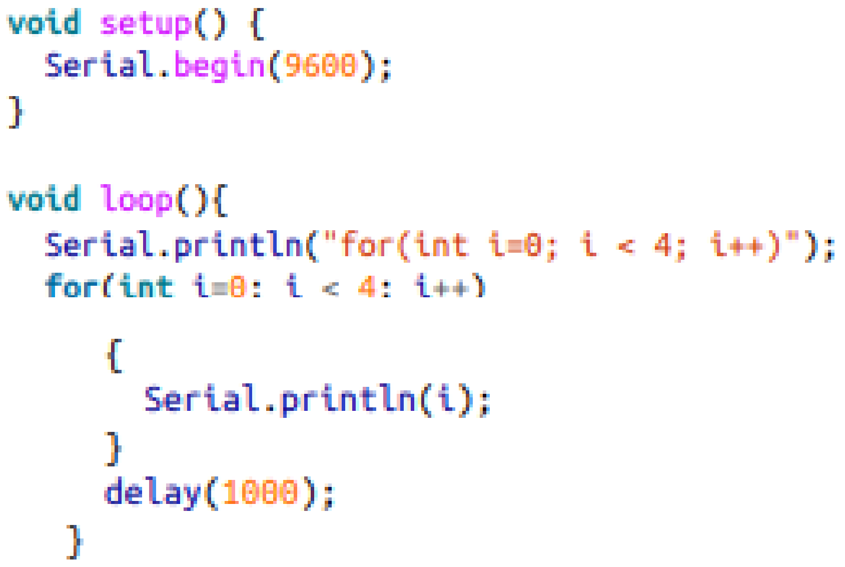
```
const int sensorPin = A0; // analog input 0

void setup()
{
  Serial.begin(9600);
  pinMode(LED_BUILTIN, OUTPUT); // enable LED pin as output
}

void loop()
{
  while(analogRead(sensorPin) > 100)
  {
    blink();    // call a function to turn an LED on and off
    Serial.print(".");
  }
  Serial.println(analogRead(sensorPin)); // this is not executed until after
```



```
}

void blink()
{
    digitalWrite(LED_BUILTIN, HIGH);
    delay(100);
    digitalWrite(LED_BUILTIN, LOW);
    delay(100);
}
```

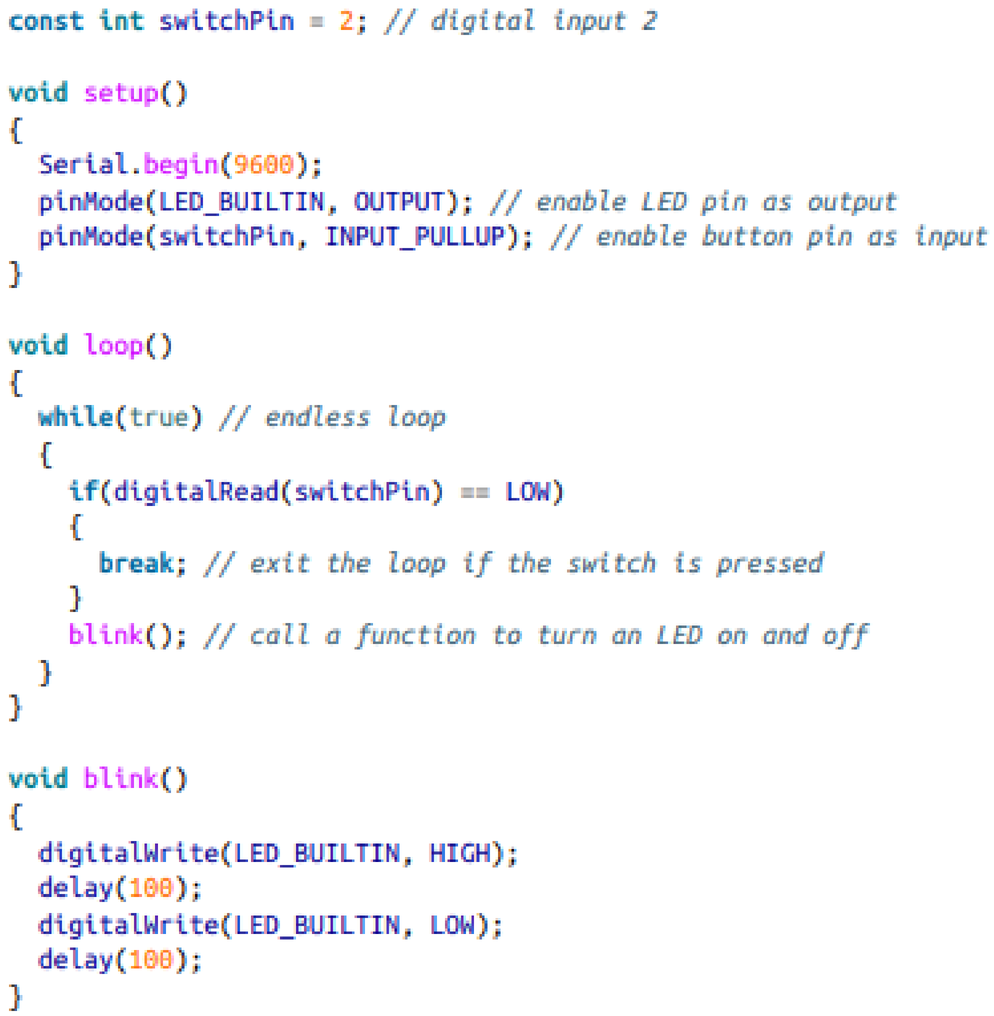
Repeating Statements with a Counter

- You want to repeat one or more statements a certain number of times. The for loop is similar to the while loop, but you have more control over the starting and ending conditions.



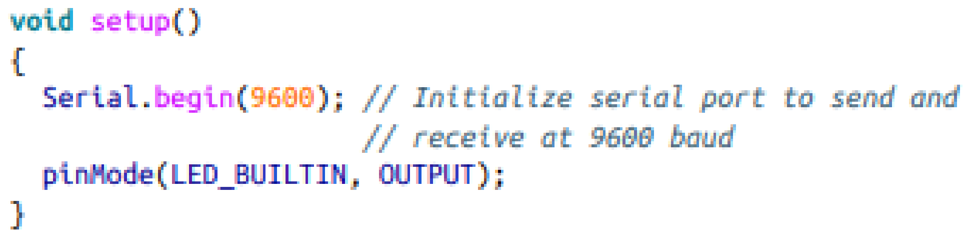
```
void setup() {
  Serial.begin(9600);
}

void loop(){
  Serial.println("for(int i=0; i < 4; i++)");
  for(int i=0; i < 4; i++)

     {
       Serial.println(i);
     }
     delay(1000);
   }
```

Breaking Out of Loops

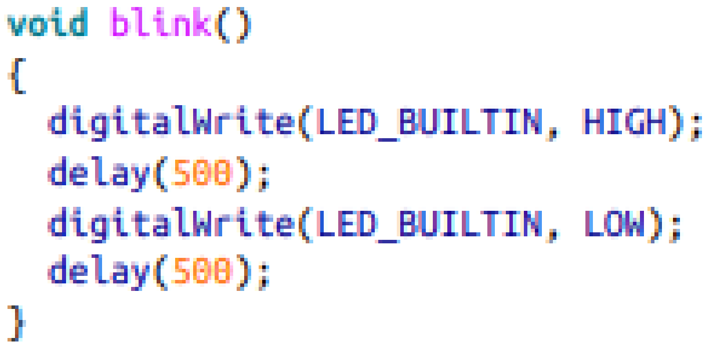


```
const int switchPin = 2; // digital input 2

void setup()
{
  Serial.begin(9600);
  pinMode(LED_BUILTIN, OUTPUT); // enable LED pin as output
  pinMode(switchPin, INPUT_PULLUP); // enable button pin as input
}

void loop()
{
  while(true) // endless loop
  {
    if(digitalRead(switchPin) == LOW)
    {
      break; // exit the loop if the switch is pressed
    }
    blink(); // call a function to turn an LED on and off
  }
}

void blink()
{
  digitalWrite(LED_BUILTIN, HIGH);
  delay(100);
  digitalWrite(LED_BUILTIN, LOW);
  delay(100);
}
```

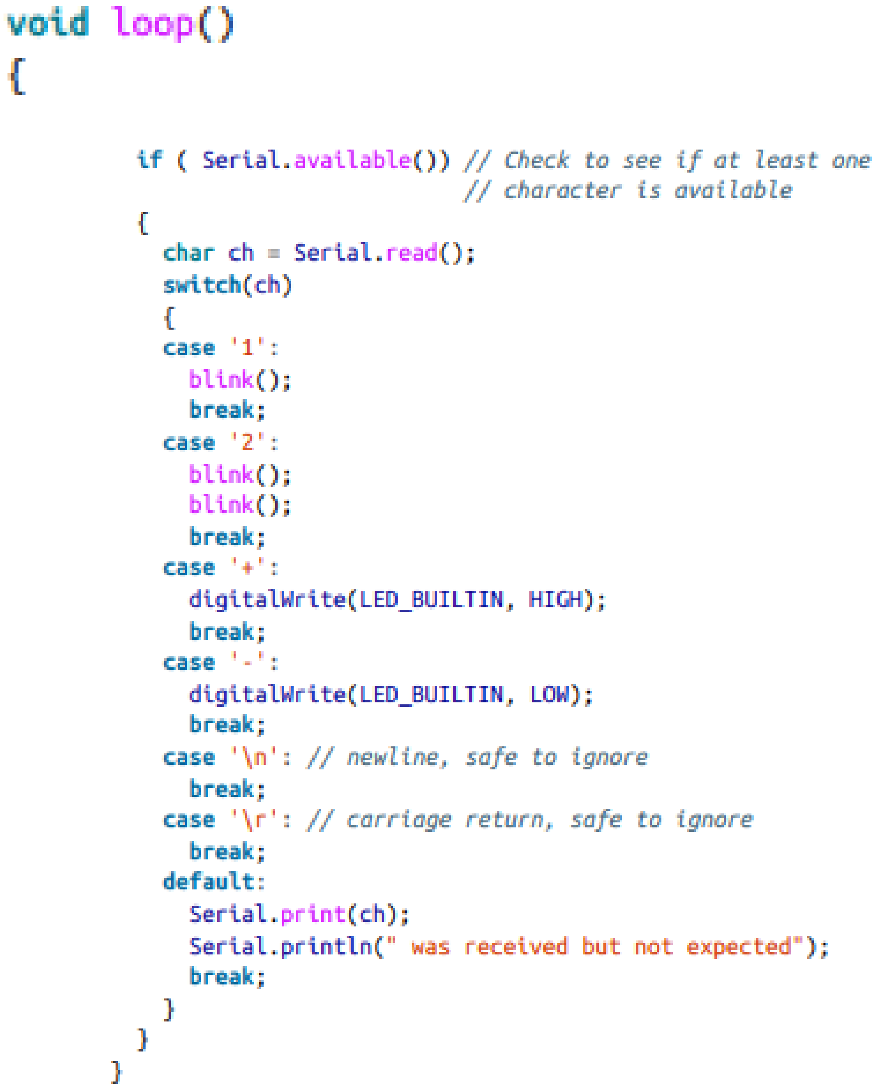
Taking a Variety of Actions Based on a Single Variable



```
void setup()
{
  Serial.begin(9600); // Initialize serial port to send and
                      // receive at 9600 baud
  pinMode(LED_BUILTIN, OUTPUT);
}
```



```
void blink()
{
  digitalWrite(LED_BUILTIN, HIGH);
  delay(500);
  digitalWrite(LED_BUILTIN, LOW);
  delay(500);
}
```



```
void loop()
{

    if ( Serial.available()) // Check to see if at least one
                             // character is available
    {
      char ch = Serial.read();
      switch(ch)
      {
      case '1':
        blink();
        break;
      case '2':
        blink();
        blink();
        break;
      case '+':
        digitalWrite(LED_BUILTIN, HIGH);
        break;
      case '-':
        digitalWrite(LED_BUILTIN, LOW);
        break;
      case '\n': // newline, safe to ignore
        break;
      case '\r': // carriage return, safe to ignore
        break;
      default:
        Serial.print(ch);
        Serial.println(" was received but not expected");
        break;
      }
    }
  }
```

Comparing Character and Numeric Values

- You want to determine the relationship between values.

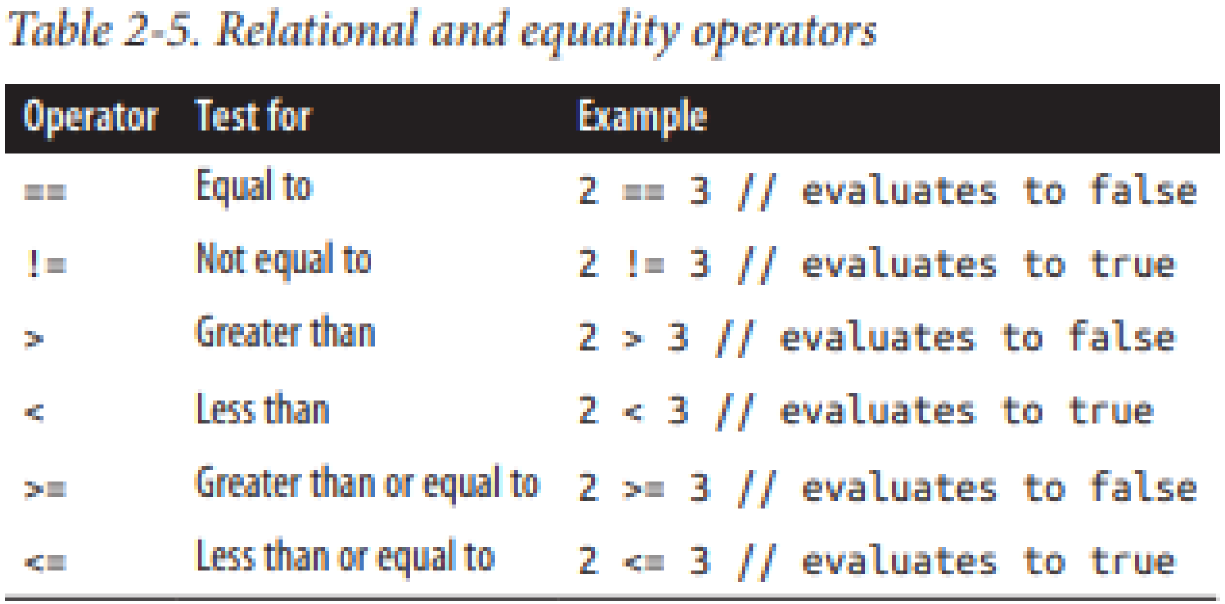
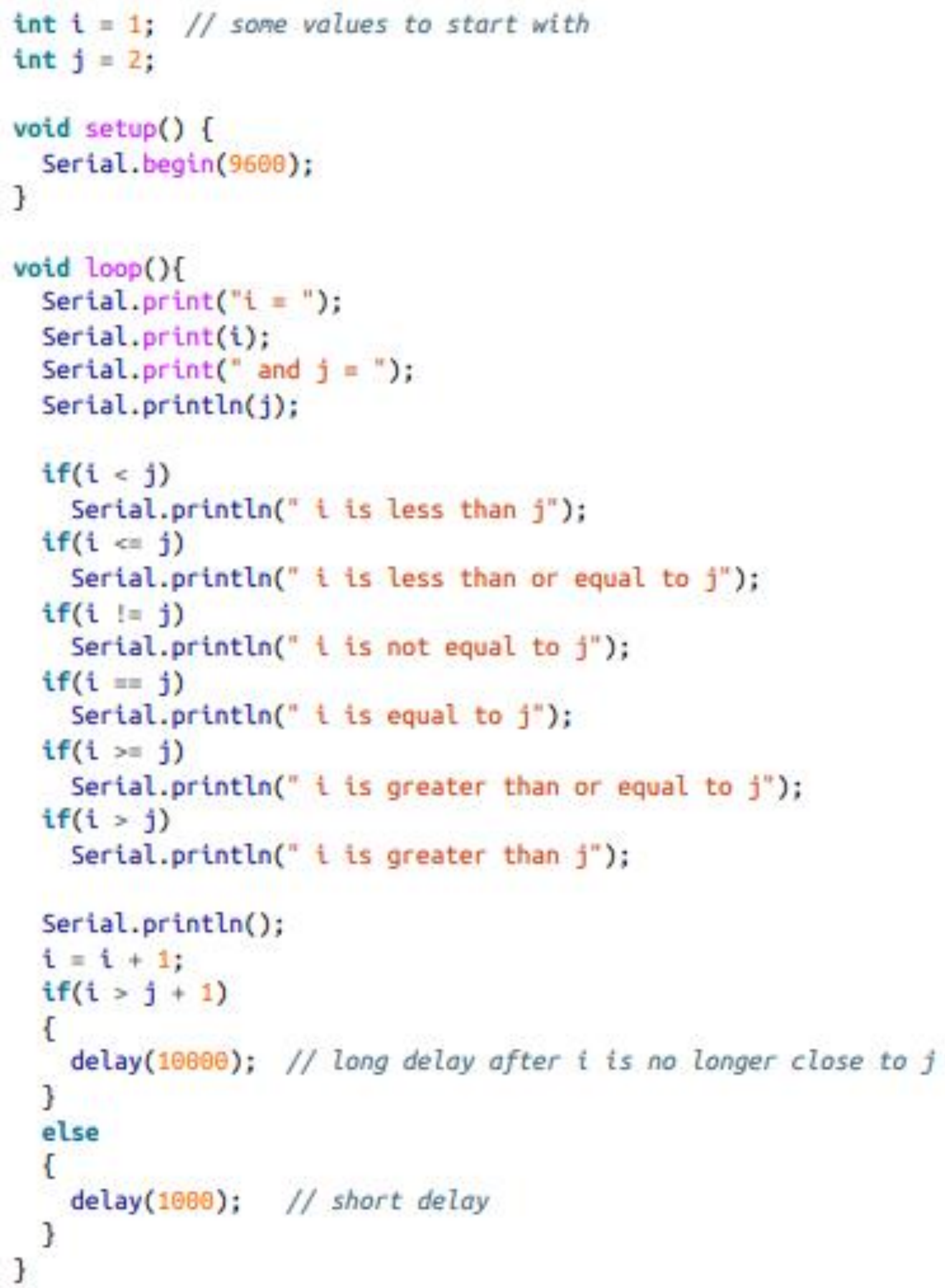


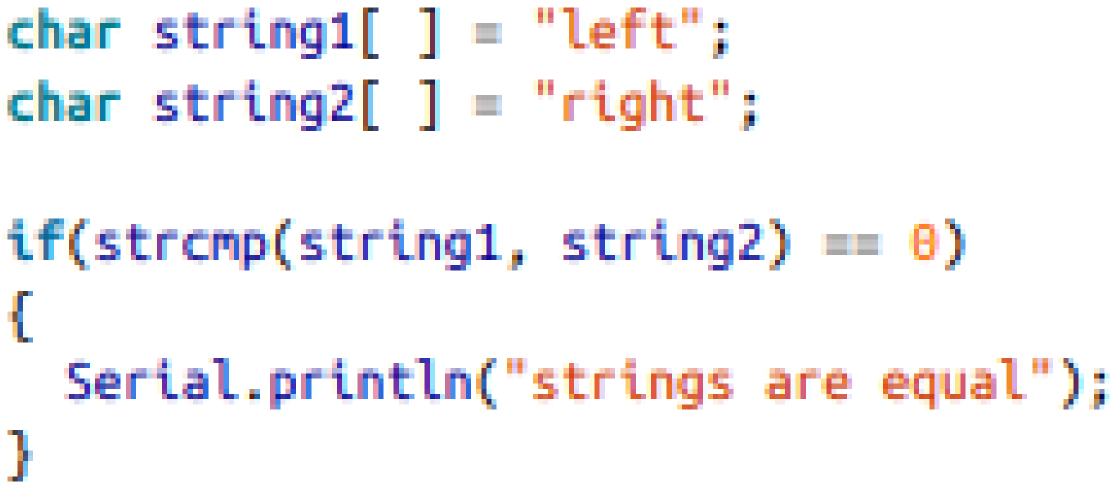
Table 2-5. Relational and equality operators

Operator	Test for	Example
`==`	Equal to	`2 == 3 // evaluates to false`
`!=`	Not equal to	`2 != 3 // evaluates to true`
`>`	Greater than	`2 > 3 // evaluates to false`
`<`	Less than	`2 < 3 // evaluates to true`
`>=`	Greater than or equal to	`2 >= 3 // evaluates to false`
`<=`	Less than or equal to	`2 <= 3 // evaluates to true`



```
int i = 1;  // some values to start with
int j = 2;

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

void loop(){
  Serial.print("i = ");
  Serial.print(i);
  Serial.print(" and j = ");
  Serial.println(j);

  if(i < j)
    Serial.println(" i is less than j");
  if(i <= j)
    Serial.println(" i is less than or equal to j");
  if(i != j)
    Serial.println(" i is not equal to j");
  if(i == j)
    Serial.println(" i is equal to j");
  if(i >= j)
    Serial.println(" i is greater than or equal to j");
  if(i > j)
    Serial.println(" i is greater than j");

  Serial.println();
  i = i + 1;
  if(i > j + 1)
  {
    delay(10000);  // long delay after i is no longer close to j
  }
  else
  {
    delay(1000);   // short delay
  }
}
```

Comparing Strings



```
char string1[ ] = "left";
char string2[ ] = "right";

if(strcmp(string1, string2) == 0)
{
  Serial.println("strings are equal");
}
```

Performing Logical Comparisons

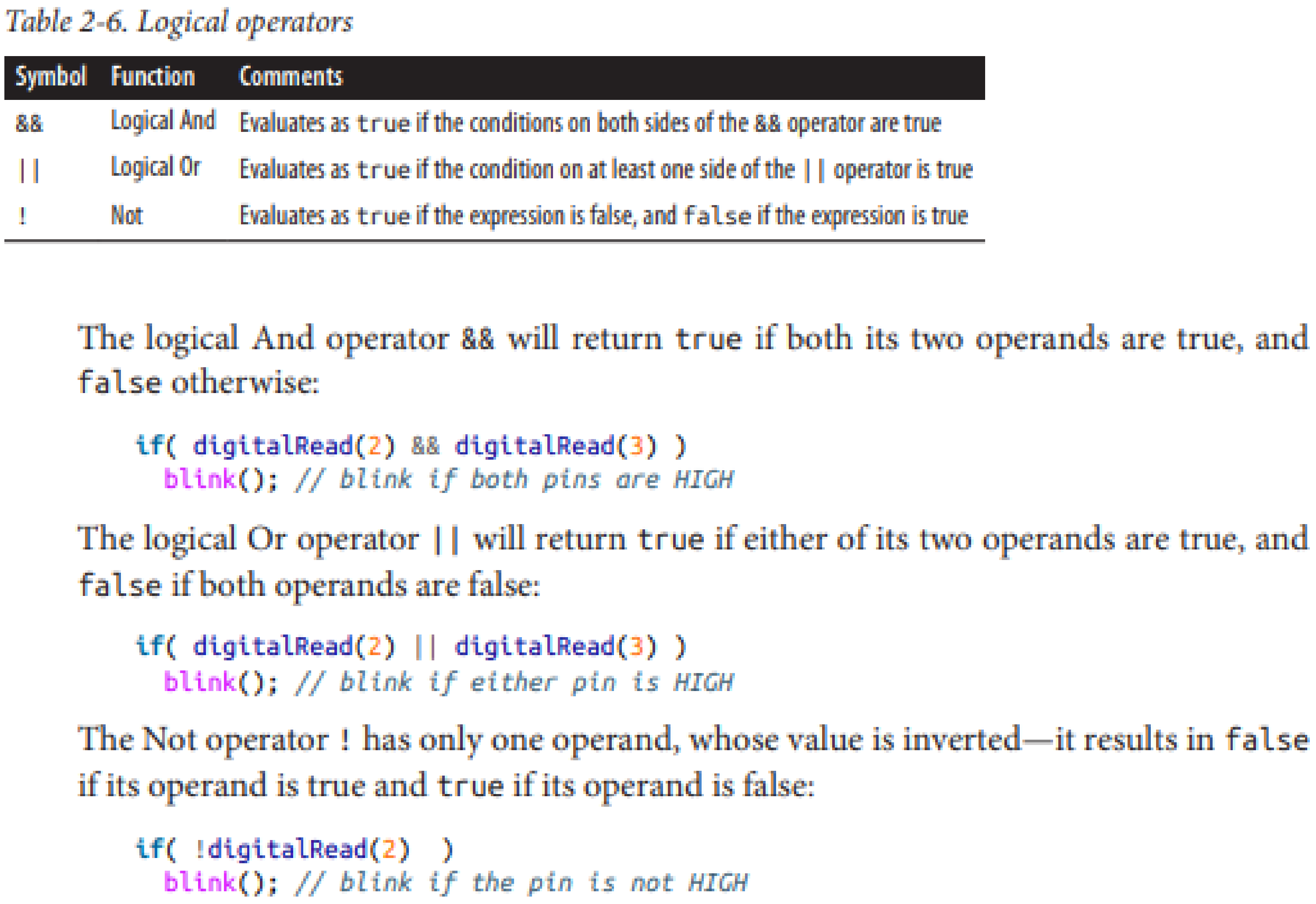


Table 2-6. Logical operators

Symbol	Function	Comments
&&	Logical And	Evaluates as `true` if the conditions on both sides of the && operator are true
||	Logical Or	Evaluates as `true` if the condition on at least one side of the || operator is true
!	Not	Evaluates as `true` if the expression is false, and `false` if the expression is true

The logical And operator `&&` will return `true` if both its two operands are true, and `false` otherwise:

```
if( digitalRead(2) && digitalRead(3) )
  blink(); // blink if both pins are HIGH
```

The logical Or operator `||` will return `true` if either of its two operands are true, and `false` if both operands are false:

```
if( digitalRead(2) || digitalRead(3) )
  blink(); // blink if either pin is HIGH
```

The Not operator `!` has only one operand, whose value is inverted—it results in `false` if its operand is true and `true` if its operand is false:

```
if( !digitalRead(2)  )
  blink(); // blink if the pin is not HIGH
```

Performing Bitwise Operations

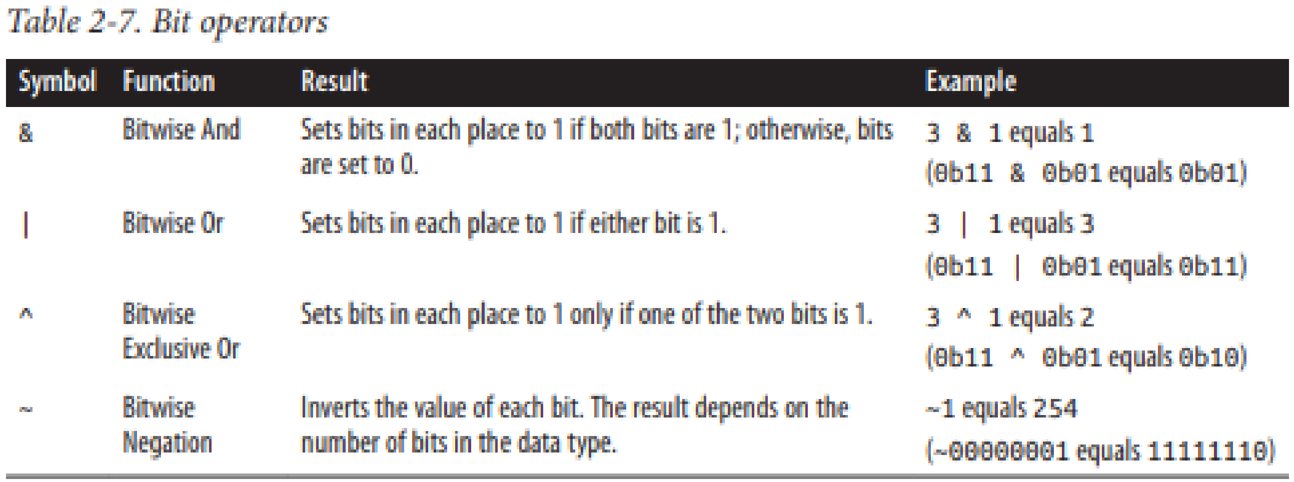
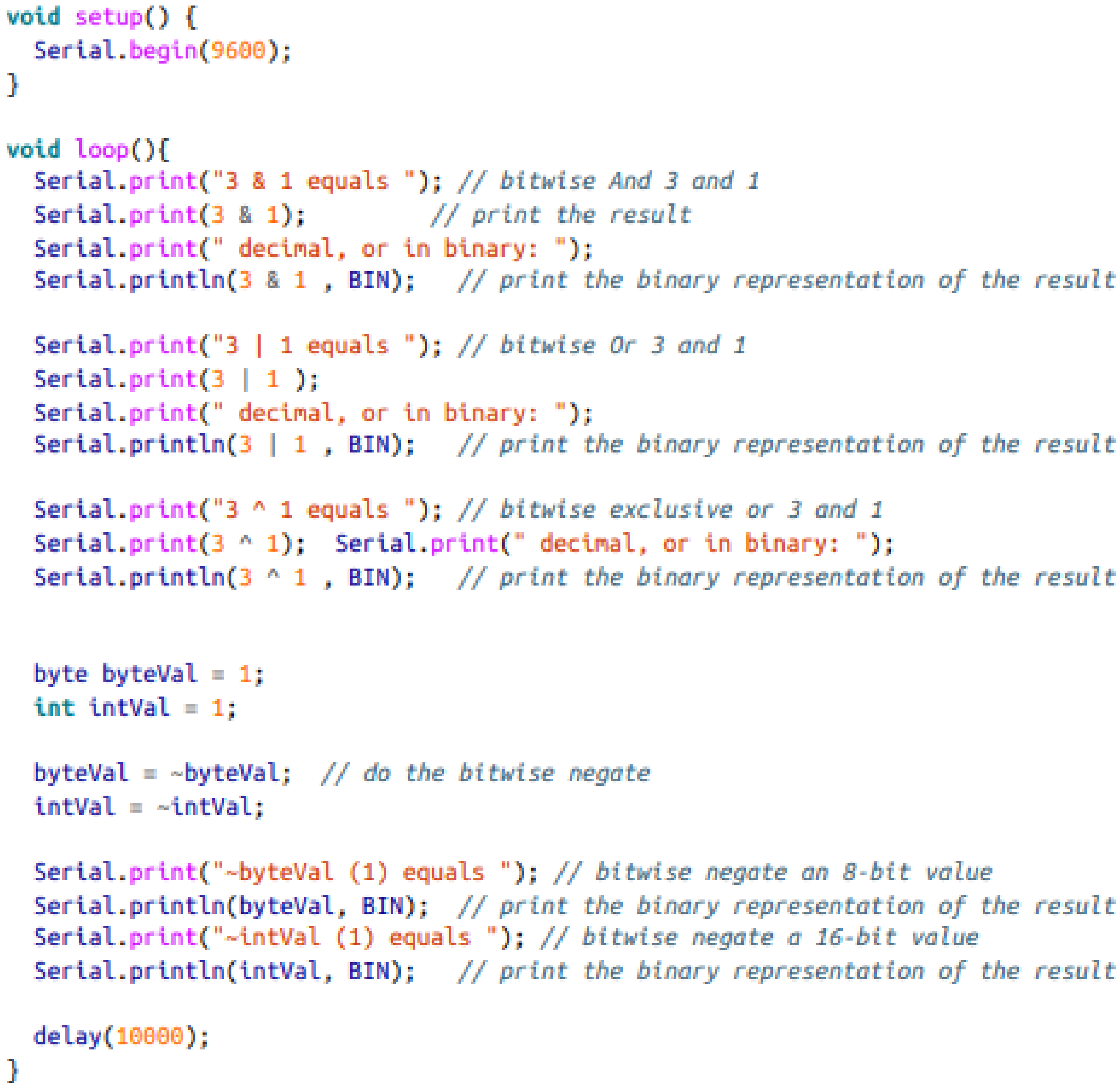


Table 2-7. Bit operators

Symbol	Function	Result	Example
&	Bitwise And	Sets bits in each place to 1 if both bits are 1; otherwise, bits are set to 0.	`3 & 1` equals `1` (`0b11 & 0b01` equals `0b01`)
|	Bitwise Or	Sets bits in each place to 1 if either bit is 1.	`3 | 1` equals `3` (`0b11 | 0b01` equals `0b11`)
^	Bitwise Exclusive Or	Sets bits in each place to 1 only if one of the two bits is 1.	`3 ^ 1` equals `2` (`0b11 ^ 0b01` equals `0b10`)
~	Bitwise Negation	Inverts the value of each bit. The result depends on the number of bits in the data type.	`~1` equals `254` (`~00000001` equals `11111110`)



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

void loop(){
  Serial.print("3 & 1 equals "); // bitwise And 3 and 1
  Serial.print(3 & 1);         // print the result
  Serial.print(" decimal, or in binary: ");
  Serial.println(3 & 1 , BIN);   // print the binary representation of the result

  Serial.print("3 | 1 equals "); // bitwise Or 3 and 1
  Serial.print(3 | 1 );
  Serial.print(" decimal, or in binary: ");
  Serial.println(3 | 1 , BIN);   // print the binary representation of the result

  Serial.print("3 ^ 1 equals "); // bitwise exclusive or 3 and 1
  Serial.print(3 ^ 1);  Serial.print(" decimal, or in binary: ");
  Serial.println(3 ^ 1 , BIN);   // print the binary representation of the result


  byte byteVal = 1;
  int intVal = 1;

  byteVal = ~byteVal;  // do the bitwise negate
  intVal = ~intVal;

  Serial.print("~byteVal (1) equals "); // bitwise negate an 8-bit value
  Serial.println(byteVal, BIN);  // print the binary representation of the result
  Serial.print("~intVal (1) equals "); // bitwise negate a 16-bit value
  Serial.println(intVal, BIN);   // print the binary representation of the result

  delay(10000);
}
```

Combining Operations and Assignment

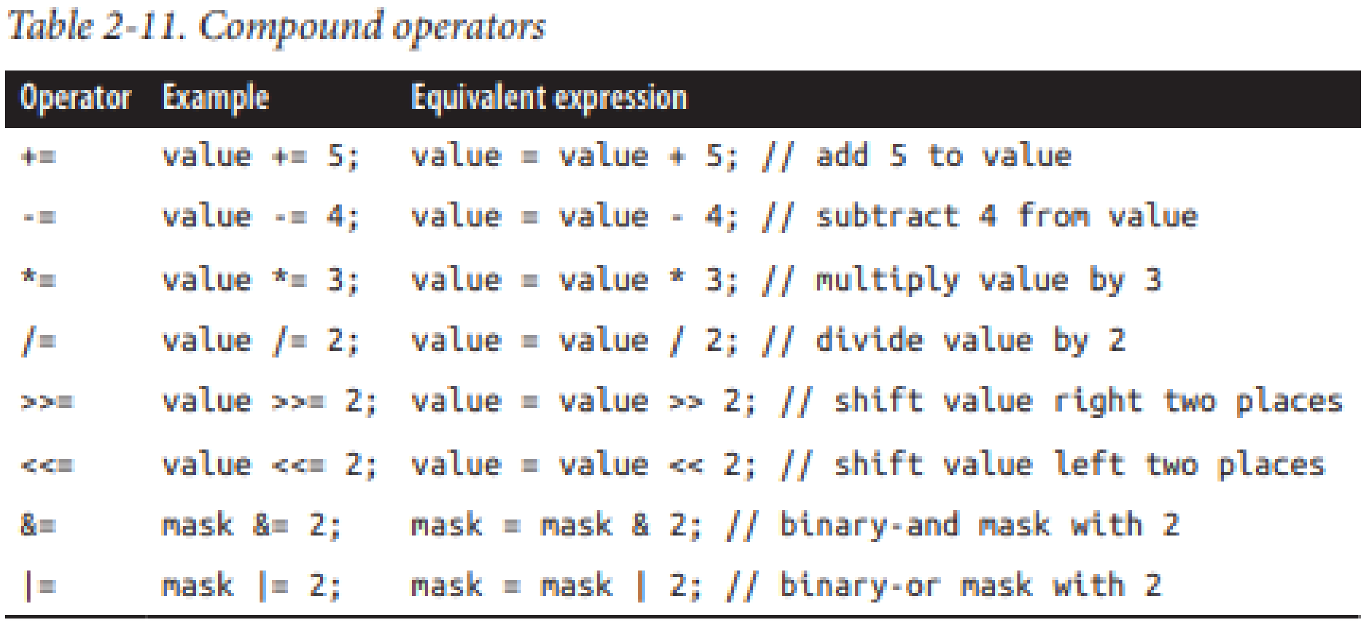


Table 2-11. Compound operators

Operator	Example	Equivalent expression
`+=`	`value += 5;`	`value = value + 5; // add 5 to value`
`-=`	`value -= 4;`	`value = value - 4; // subtract 4 from value`
`*=`	`value *= 3;`	`value = value * 3; // multiply value by 3`
`/=`	`value /= 2;`	`value = value / 2; // divide value by 2`
`>>=`	`value >>= 2;`	`value = value >> 2; // shift value right two places`
`<<=`	`value <<= 2;`	`value = value << 2; // shift value left two places`
`&=`	`mask &= 2;`	`mask = mask & 2; // binary-and mask with 2`
`|=`	`mask |= 2;`	`mask = mask | 2; // binary-or mask with 2`