4.1. Values and Data Types

Programs may be thought of as being made up of two things:

1. Data
2. Operations that manipulate data

This chapter focuses primarily on the first of these two fundamental components, data.

Data can be stored in a program in a variety of ways. The most basic unit of data is a value.

A **value** is a specific piece of data, such as a word or a number. Some examples are `5`, `5.2`, and `"Hello, World!"`.

Each value belongs to a category called a **data type**. We will see many different data types throughout the course, the first two of which are the **number** and **string** types. Numeric values such as `4` and `3.3` are a numbers. Sequences of characters enclosed in quotes, such as `"Hello, World!"`, are strings, so-called because they contain a string of letters. Strings must be enclosed in either single or double quotes.

If you are not sure what data type a value falls into, precede the value with `typeof`.

Example

```
console.log(typeof "Hello, World!");
console.log(typeof 17);
console.log(typeof 3.14);
```

Console Output

```
string
number
number
```

Not surprisingly, JavaScript reports that the data type of `"Hello, World!"` is `string`, while the data type of both `17` and `3.14` is `number`. Note that some JavaScript environments may print type names and strings with single quotes around them, as in `'string'`, `'number'`, and `'hello'`.

Note

Notice that `console.log(typeof "Hello, World!");` prints out `string` to the console. The `typeof` keyword is not printed to the console because the statement `typeof "Hello, World!"` is an **expression**. Briefly, expressions are code segments that are reduced to a value. We will learn more about expressions soon.

Note

`typeof` is a JavaScript entity known as an **operator**. It is similar to a function in that it carries out some kind of action, though the syntax is different from that of functions (notice using `typeof` does not require parenthses).

There are data types other than string and number, including object and function, which we will learn about in future chapters.

4.1.1. More On Strings

What about values like `"17"` and `"3.2"`? They look like numbers, but they are in quotation marks like strings.

Run the following code to find out.

Try It!

```
console.log(typeof "17");
console.log(typeof "3.2");
```

repl.it

Question

What is the data type of the values `"17"` and `"3.2"`?

Strings in JavaScript can be enclosed in either single quotes (`'`) or double quotes (`"`).

Example

```
console.log(typeof 'This is a string');
console.log(typeof "And so is this");
```

Console Output

```
string
string
```

Double-quoted strings can contain single quotes inside them, as in `"Bruce's beard"`, and single quoted strings can have double quotes inside them, as in `'The knights who say "Ni!"'`.

JavaScript doesn't care whether you use single or double quotes to surround your strings. Once it has parsed the text of your program or command, the way it stores the value is identical in all cases, and the surrounding quotes are not part of the value.

Warning

If a string contains a single quote (such as `"Bruce's beard"`) then surrounding it with single quotes gives unexpected results.

```
console.log('Bruce's beard');
```

4.1.2. More On Numbers

When you type a large integer value, you might be tempted to use commas between groups of three digits, as in `42,000`. This is not a legal integer in JavaScript, but it does mean something else, which is legal:

Example

```
console.log(42000);
console.log(42,000);
```

Console Output

```
42000
42 0
```

Well, that's not what we expected at all! Because of the comma, JavaScript chose to treat `42,000` as a *pair* of values. In fact, the `console.log` function can print any number of values as long as you separate them by commas. Notice that the values are separated by spaces when they are displayed.

Example

```
console.log(42, 17, 56, 34, 11, 4.35, 32);
```

```
    2console.log(3.4, "hello", 45);
```

Console Output

```
42 17 56 34 11 4.35 32
3.4 'hello' 45
```

Remember not to put commas or spaces in your integers, no matter how big they are. Also revisit what we said in the chapter How Programs Work: formal languages are strict, the notation is concise, and even the smallest change might mean something quite different from what you intend.

4.1.3. Type Systems

Every programming language has a **type system**, which is the set of rules that determine how the languages deals with data of different types. In particular, how values are divided up into different data types is one characteristic of a type system.

In many programming languages, integers and floats are considered to be different data types. For example, in Python `42` is of the `int` data type, while `42.0` is of the `float` data type.

Note

While JavaScript does not distinguish between floats and integers, at times we may wish to do so in our programs. For example, an inventory-tracking program stores items and the number of each number in stock. Since a store cannot have 3.5 shirts in stock, the programmer makes the quantity of each item integer values as opposed to floats.

When discussing the differences between programming languages, the details of type systems are one of the main factors that programmers consider. There are other aspects of type systems beyond just how values are categorized. We will explore these in future lessons.

4.1.4. Check Your Understanding

Question

Which of these is *not* a data type in JavaScript?

a. number
b. string
c. letter
d. object

4.2. Type Conversion

Sometimes it is necessary to convert values from one type to another. A common example is when a program receives input from a user or a file. In this situation, numeric data may be passed to the program as strings.

JavaScript provides a few simple functions that will allow us to convert values to different data types. The functions `Number` and `String` will (attempt to) convert their arguments into types `number` and `string`, respectively. We call these **type conversion** functions.

The `Number` function can take a string and turn it into an integer. Let us see this in action:

Example

```
console.log(Number("2345"));
console.log(typeof Number("2345"))
console.log(Number(17));
```

Console Output

```
2345
number
17
```

What happens if we attempt to convert a string to a number, and the string doesn't directly represent a number?

Example

```
console.log(Number("23bottles"));
```

Console Output

```
NaN
```

This example shows that a string has to be a syntactically legal number for conversion to go as expected. Examples of such strings are `"34"` or `"-2.5"`. If the value cannot be cleanly converted to a number then `NaN` will be returned, which stands for "not a number."

Note

`NaN` is a **special value** in JavaScript that represents that state of not being a number. We will learn more about `NaN` and other special values in a later chapter.

The type conversion function `String` turns its argument into a string. Remember that when we print a string, the quotes may be removed. However, if we print the type, we can see that it is definitely `'string'`.

Example

```
console.log(String(17));
console.log(String(123.45));
console.log(typeof String(123.45));
```

Console Output

```
17
123.45
string
```

4.2.1. Check Your Understanding

Question

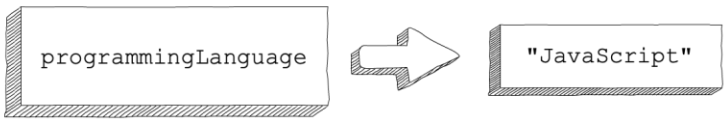
Which of the following strings result in `NaN` when passed to `Number`? (Feel free to try running each of the conversions.)

a. `'3'`
b. `'three'`
c. `'3 3'`
d. `'33'`

4.3. Variables

One of the most powerful features of a programming language is the ability to manipulate variables. A **variable** is a name that refers to a value. Recall that a value is a single, specific piece of data, such as a specific number or string. Variables allow us to store values for later use.

A useful visual analogy for how a variable works is that of a label that *points to* a piece of data.



A variable can be visualized as a label pointing to a specific piece of data.

In this figure, the name `programmingLanguage` points to the string value `"JavaScript"`. This is more than an analogy, since it also is representative of how a variable and its value are stored in a computer's memory.

With this analogy in mind, let's look at how we can formally create variables in JavaScript.

4.3.1. Declaring and Initializing Variables With `let`

To create a variable in JavaScript, create a new name for the variable and precede it with the keyword `let`:

```
let programmingLanguage;
```

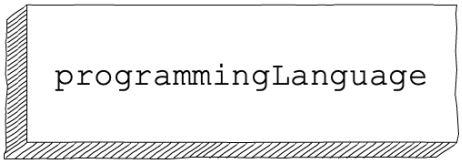
This creates a variable named `programmingLanguage`. The act of creating a variable is referred to as **variable declaration**, or simply **declaration**.

Once a variable has been declared, it may be given a value using an **assignment statement**, which uses `=` to give a variable a value.

```
let programmingLanguage;
programmingLanguage = "JavaScript";
```

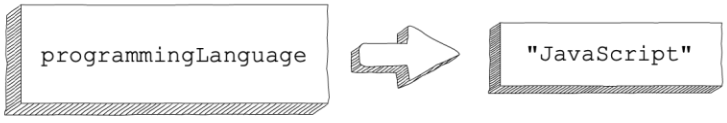
The act of assigning a variable a value for the first time is called **initialization**.

The first line creates a variable that does not yet have a value. The variable is a label that does not point to any data.



The result of `let programmingLanguage;`

The second line assigns the variable a value, which connects the name to the given piece of data.



The result of `programmingLanguage = "JavaScript";`

It is possible to declare *and* initialize a variable with a single line of code. This is the most common way to create a variable.

```
let programmingLanguage = "JavaScript";
```

Warning

You will see some programmers use `var` to create a variable in JavaScript, like this:

```
var programmingLanguage = "JavaScript";
```

While this is valid syntax, you should NOT use `var` to declare a variable. Using `var` is old JavaScript syntax, and it differs from `let` in important ways that we will learn about later. When you see examples using `var`, use `let` instead.

If you're curious, read about the differences between var and let.

To give a variable a value, use the **assignment operator**, `=`. This operator should not be confused with the concept of *equality*, which expresses whether two things are the "same" (we will see later that equality uses the `===` operator). The assignment statement links a *name*, on the left-hand side of the operator, with a *value*, on the right-hand side. This is why you will get an error if you try to run:

```
"JavaScript" = programmingLanguage;
```

An assignment statement must have the name on the left-hand side, and the value on the right-hand side.

Tip

To avoid confusion when reading or writing code, say to yourself:

`programmingLanguage` is assigned `'JavaScript'`

or

`programmingLanguage` gets the value `'JavaScript'`.

Don't say:

`programmingLanguage` equals `'JavaScript'`.

Warning

What if, by mistake, you leave off `let` when declaring a variable?

```
programmingLanguage = "JavaScript";
```

Contrary to what you might expect, JavaScript will not complain or throw an error. In fact, creating a variable without `let` is valid syntax, but it results in very different behavior. Such a variable will be a **global variable**, which we will discuss later.

The main point to keep in mind for now is that you should *always* use `let` unless you have a specific reason not to do so.

4.3.2. Evaluating Variables

After a variable has been created, it may be used later in a program anywhere a value may be used. For example, `console.log` prints a value, we can also give `console.log` a variable.

Example

These two examples have the exact same same output.

```
console.log("Hello, World!");
```

```
let message = "Hello, World!";
console.log(message);
```

```
2console.log(message);
```

When we refer to a variable name, we are **evaluating** the variable. The effect is just as if the value of the variable is substituted for the variable name in the code when executed.

Example

```
1let message = "What's up, Doc?";
2let n = 17;
3let pi = 3.14159;
4
5console.log(message);
6console.log(n);
7console.log(pi);
```

Console Output

```
What's up, Doc?
17
3.14159
```

In each case, the printed result is the value of the variable.

Like values, variables also have types. We determine the type of a variable the same way we determine the type of a value, using `typeof`.

Example

```
1let message = "What's up, Doc?";
2let n = 17;
3let pi = 3.14159;
4
5console.log(typeof message);
6console.log(typeof n);
7console.log(typeof pi);
```

Console Output

```
string
number
number
```

The type of a variable is the type of the data it currently refers to.

4.3.3. Reassigning Variables

We use variables in a program to “remember” things, like the current score at the football game. As their name implies, variables can change over time, just like the scoreboard at a football game. You can assign a value to a variable, and later assign it a different value.

To see this, read and then run the following program in a code editor. You’ll notice that we change the value of `day` three times, and on the third assignment we even give it a value that is of a different data type.

```
1 let day = "Thursday";
2 console.log(day);
3
4 day = "Friday";
5 console.log(day);
6
7 day = 21;
```

```
8 console.log(day);
```

A great deal of programming involves asking the computer to remember things. For example, we might want to keep track of the number of missed calls on your phone. Each time another call is missed, we can arrange to update a variable so that it will always reflect the correct total of missed calls.

Note

We only use `let` when *declaring* a variable, that is, when we create it. We do NOT use `let` when reassigning the variable to a different value. In fact, doing so will result in an error.

4.3.4. Check Your Understanding

Question

What is printed when the following code executes?

```
1 let day = "Thursday";
2 day = 32.5;
3 day = 19;
4 console.log(day);
```

a. Nothing is printed. A runtime error occurs.
b. `Thursday`
c. `32.5`
d. `19`

Question

How can you determine the type of a variable?

a. Print out the value and determine the data type based on the value printed.
b. Use `typeof`.
c. Use it in a known equation and print the result.
d. Look at the declaration of the variable.

Question

Which line is an example of variable initialization? (*Note: only one line is such an example.*)

```
1 let a;
2 a = 42;
3 a = a + 3;
```

4.4. More On Variables

The previous section covered creating, evaluating, and reassigning variables. This section will cover some additional, more nuanced topics related to variables.

4.4.1. Creating Constants With `const`

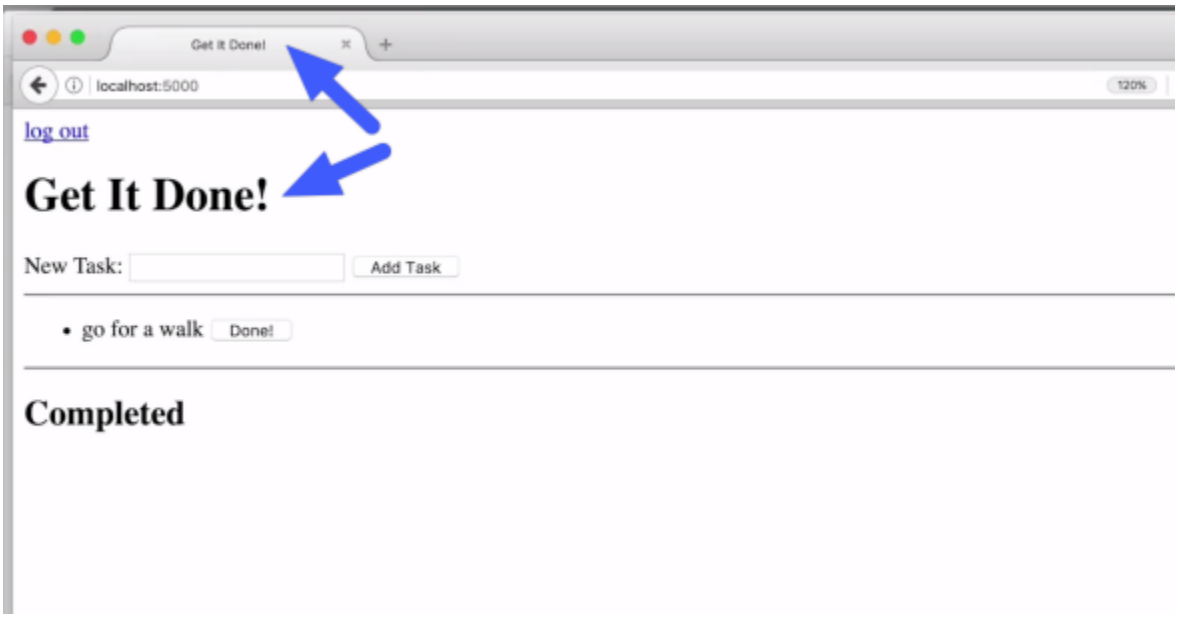
One of the key features of variables that we have discussed so far is their ability to change value. We can create a variable with one value, and then reassign it to another value.

```
let programmingLanguage = "JavaScript";
programmingLanguage = "Python";

```

In some situations, we want to create variables that cannot change value. Many programming languages, including JavaScript, provide mechanisms for programmers to make variables that are constant.

For example, suppose that we are writing a to-do list web application, named "Get It Done!" The title of the application might appear in multiple places, such as the title bar and the main page header.



An example to-do list web application

We might store the name of our application in a variable so that it can be referenced anywhere we want to display the application name.

```
let appName = "Get It Done!";
```

This allows us to simply refer to the `appName` variable any time we want to use it throughout our application. If we change the name of the application, we only have to change one line of code, where the `appName` variable is initialized.

One problem with this approach is that an unwitting programmer might change the value of `appName` later in the code, leading to inconsistent references to the application name. In other words, the title bar and main page header could reference different names.

Using `const` rather than `let` to create a variable ensures that the value of the declared variable cannot be changed.

```
const appName = "Get It Done!";
```

Such an unchangeable variable is known as a **constant**, since its value is just that.

How does JavaScript prevent a programmer from changing the value of a constant? Let's find out. Try running the following code in an editor. What happens?

Example

```
 const appName = "Get It Done";
1appName = "Best TODO application Ever!";
2
```

Console Output

```
TypeError: Assignment to constant variable.
```

As we've seen with other examples—such as trying to declare a variable twice, using incorrect syntax, or failing to enclose strings in quotes—JavaScript prevents undesired code from executing by throwing an error.

4.4.2. Naming Variables

4.4.2.1. Valid Variable Names

As you may have discovered already, not just any sequence of characters is a valid variable name. For example, if we try to declare a variable with a name containing a space, JavaScript complains.

Example

```
let application name;
```

Console Output

```
SyntaxError: Unexpected identifier
```

In this case, "identifier" is another term for variable name, so the error message is saying that the variable name is not valid, or is "unexpected".

JavaScript provides a broad set of rules for naming variables, but there is no reason to go beyond a few easy-to-remember guidelines:

1. Use only the characters 0-9, a-z, A-Z, and underscore. In other words, do not use special characters or whitespace (space, tab, and so on).
2. Do not start a variable name with a number.
3. Avoid starting a variable name with an underscore. Doing so is a convention used by some JavaScript developers to mean something very specific about the variable, and should be avoided.
4. Do not use **keywords**, which are words reserved by JavaScript for use by the language itself. We'll discuss these in detail in a moment.

Following these guidelines will prevent you from creating illegal variable names. While this is important, we should also strive to create good variable names.

4.4.2.2. Good Variable Names

Writing good code is about more than writing code that simply works and accomplishes the task at-hand. It is also about writing code that can be read, updated, and maintained as easily as possible. How to write code that achieves these goals is a theme we will return to again and again.

One of the primary ways that code can be written poorly is by using bad variable names. For example, consider the following program. While we haven't introduced each of the components used here, you should be able to come to a general understanding of the new components.

```
let x = 5;
const y = 3.14;
let z = y * x ** 2;
console.log(z);

```

Understanding what this program is trying to do is not obvious, to say the least. The main problem is that the variable names `x`, `y`, and `z` are not descriptive. They don't tell us anything about what they represent, or how they will be used.

Variable names should be descriptive, providing context about the data they contain and how they will be used.

Let's look at an improved version this program.

```
let radiusOfCircle = 5;
const pi = 3.14;
let areaOfCircle = pi * radiusOfCircle ** 2;
console.log(areaOfCircle);

```

With improved variable names, it now becomes clear that the program is calculating the area of a circle of radius 5.

Tip

When considering program readability, think about whether or not your code will make sense to another programmer. It is not enough for code to be readable by only the programmer that originally wrote it.

4.4.2.3. Camel Case Variable Names

There is one more aspect of naming variables that you should be aware of, and that is conventions used by professional programmers. Conventions are not formal rules, but are informal practices adopted by a group.

Example

In the United States, it is common for two people to greet each other with a handshake. In other countries and cultures, such as some in east Asia, the conventional greeting is to bow.

Failing to follow a social convention is not a violation of the law, but is considered impolite nonetheless. It is a signal that you are not part of the group, or do not respect its norms.
There are a variety of types of conventions used by different groups of programmers. One common type of convention is that programmers that specialize in a specific language will adopt certain variable naming practices.

In JavaScript, most programmers use the **camel case** style, which stipulates that variable names consist of names or phrases that:

- are joined together to omit spaces,
- start with a lowercase letter, and
- capitalize each internal word.

In the example from the previous section, the descriptor “area of circle” became the variable name `areaOfCircle`. This convention is called camel case because the capitalization of internal words is reminiscent of a camel’s humps. Another another common name for this convention is **lower camel case**, since names start with a lowercase letter.

Note

Different programming languages often have different variable-naming conventions. For example, in Python the convention is to use all lowercase letters and separate words with underscores, as in `area_of_circle`.
We will use the lower camel case convention throughout this course, and strongly encourage you to do so as well.

4.4.3. Keywords

Our last note on naming variables has to do with a collection of words that are reserved for use by the JavaScript language itself. Such words are called **keywords**, or **reserved words**.

Any word that is formally part of the JavaScript language syntax is a keyword. So far, we have seen only four keywords: `let`, `const`, `var`, and `typeof`.

Warning

While `console` and `console.log` may seem like keywords, they are actually slightly different things. They are entities (an object and a function, respectively) that are available by default in most JavaScript environments. Attempting to use a keyword for anything other than it’s intended use will result in an error. To see this, let’s try to name a variable `const`.

Example

```
let const;
```

Console Output

```
let const
^^^^^

SyntaxError: Unexpected token const
```

Tip

Most code editors will highlight keywords in a different color than variables or other parts of your code. This serves as a visual cue that a given word is a keyword, and can help prevent mistakes.
We will not provide the full list of keywords at this time, but rather point them out as we learn about each of them. If you are curious, the full list is available at MDN.

4.4.4. Check Your Understanding

Question

Which is the best keyword for declaring a variable in most situations?

a. `var`
b. `let`
c. `const`
d. (no keyword)

4.5. Expressions and Evaluation

An **expression** is a combination of values, variables, operators, and calls to functions. An expression can be thought of as a formula that is made up of multiple pieces.

The *evaluation* of an expression produces a value, known as the **return value**. We say that an expression **returns** a value.

Expressions need to be evaluated when the code executes in order to determine the return value, or specific piece of data that should be used. Evaluation is the process of computing the return value.

If you ask JavaScript to print an expression using `console.log`, the interpreter **evaluates** the expression and displays the result.

Example

```
console.log(1 + 1);
```

Console Output

```
2
```

This code prints not `1 + 1` but rather the *result* of calculating `1 + 1`. In other words, `console.log(1 + 1)` prints the value `2`. This is what we would expect.

Since evaluating an expression produces a value, expressions can appear on the right-hand side of assignment statements.

Example

```
let sum = 1 + 2;
console.log(sum);
```

Console Output

```
3
```

The value of the variable `sum` is the result of evaluating the expression `1 + 2`, so the value `3` is printed.

A value all by itself is a simple expression, and so is a variable. Evaluating a variable gives the value that the variable refers to. This means that line 2 of the example above also contains the simple expression `sum`.

4.6. Operations

4.6.1. Operators and Operands

Now that we can store data in variables, let's explore how we can generate new data from existing data.

An **operator** is one or more characters that represents a computation like addition, multiplication, or division. The values an operator works on are called **operands**.

The following are all legal JavaScript expressions whose meaning is more or less clear:

- `20 + 32`
- `hour - 1`
- `hour * 60 + minute`
- `minute / 60`
- `5 ** 2`
- `(5 + 9) * (15 - 7)`

For example, in the calculation `20 + 32`, the operator is `+` and the operands are `20` and `32`.

The symbols `+` and `-`, and the use of parentheses for grouping, mean in JavaScript what they mean in mathematics. The asterisk (`*`) is the symbol for multiplication, and `**` is the symbol for exponentiation. Addition, subtraction, multiplication, and exponentiation all do what you expect.

Example

```
console.log(2 + 3);
console.log(2 - 3);
console.log(2 * 3);
console.log(2 ** 3);
console.log(3 ** 2);
```

Console Output

```
5
-1
6
8
9
```

We use the same terminology as before, stating that `2 + 3` **returns** the value `5`.

When a variable name appears in the place of an operand, it is replaced with the value that it refers to before the operation is performed. For example, suppose that we wanted to convert 645 minutes into hours. Division is denoted by the operator `/`.

Example

```
let minutes = 645;
let hours = minutes / 60;
console.log(hours);
```

Console Output

```
10.75
```

In summary, operators and operands can be combined to create expressions that are evaluated upon execution. Let's discuss some specific types of operators

4.6.2. Arithmetic Operators

Some of most commonly-used operators are the **arithmetic operators**, which carry out basic mathematical operations. These behave exactly as you are used to, though the modulus operator (`%`) may be new to you.

Arithmetic operators

Operator	Description	Example
Addition (`+`)	Adds the two operands	`2 + 3` returns `5`
Subtraction (`-`)	Subtracts the two operands	`2 - 3` returns `-1`
Multiplication (`*`)	Multiplies the two operands	`2 * 3` returns `6`
Division (`/`)	Divides the first operand by the second	`6 / 2` returns `3`
Modulus (`%`)	Aka the remainder operator. Returns the integer remainder of dividing the two operands.	`7 % 5` returns `2`
Exponentiation (`**`)	Calculates the base (first operand) to the exponent (second operand) power, that is, base^{exponent}	`3 ** 2` returns `9` `5 ** -1` returns `0.2`
Increment (`++`)	Adds one to its operand. If used before the operand (`++x`), returns the value of its operand after adding one; if used after the operand (`x++`), returns the value of its operand before adding one.	If `x` is `2`, then `++x` sets `x` to `3` and returns `3`, whereas `x++` returns `2` and, only then, sets `x` to `3`
Decrement (`--`)	Subtracts one from its operand. The return value is analogous to that for the increment operator.	If `x` is `2`, then `--x` sets `x` to `1` and returns `1`, whereas `x--` returns `2` and, only then, sets `x` to `1`

While the modulus operator (`%`) is common in programming, it is not used much outside of programming. Let's explore how it works with a few examples.

The `%` operator returns the *remainder* obtained by carrying out integer division of the first operand by the second operand. Therefore, `5 % 3` is `2` because 3 goes into 5 one whole time, with a remainder of 2 left over.

Examples

- 12 % 4 is 0, because 4 divides 12 evenly (that is, there is no remainder)

- 13 % 7 is 6
- 6 % 2 is 0
- 7 % 2 is 1

The last two examples illustrate a general rule: An integer x is even exactly when x % 2 is 0 and is odd exactly when x % 2 is 1.

Note

The value returned by `a % b` will be in the range from `0` to `b` (not including `b`).

Tip

If remainders and the modulus operator seem tricky to you, we recommend getting additional practice at Khan Academy.

4.6.3. Order of Operations

When more than one operator appears in an expression, the order of evaluation depends on the **rules of precedence**. JavaScript follows the same precedence rules for its arithmetic operators that mathematics does.

1. Parentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, `2 * (3 - 1)` is 4, and `(1 + 1) ** (5 - 2)` is 8. You can also use parentheses to make an expression easier to read, as in `(minute * 100) / 60`, even though it doesn't change the result.
2. Exponentiation has the next highest precedence, so `2 ** 1 + 1` is 3 and not 4, and `3 * 1 ** 3` is 3 and not 27. Can you explain why?
3. Multiplication, division, and modulus operators have the same precedence, which is higher than addition and subtraction, which also have the same precedence. So `2 * 3 - 1` yields 5 rather than 4, and `5 - 2 * 2` is 1, not 6.
4. Operators with the *same* precedence are evaluated from left-to-right. So in the expression `6 - 3 + 2`, the subtraction happens first, yielding 3. We then add 2 to get the result 5. If the operations had been evaluated from right to left, the result would have been `6 - (3 + 2)`, which is 1.

Tip

The acronym PEMDAS can be used to remember order of operations:

P = parentheses

E = exponentiation

M = multiplication

D = division

A = addition

S = subtraction

Note

Due to an historical quirk, an exception to the left-to-right rule is the exponentiation operator `**`. A useful hint is to always use parentheses to force exactly the order you want when exponentiation is involved:

```
1// the right-most ** operator is applied first
2console.log(2 ** 3 ** 2)
3
4// use parentheses to force the order you want
```

```
    5console.log((2 ** 3) ** 2)
```

Console Output

```
512
64
```

4.6.4. Check Your Understanding

Question

What is the value of the following expression?

```
16 - 2 * 5 / 3 + 1
```

a. 14
b. 24
c. 3
d. 13.666666666666666

Question

What is the output of the code below?

```
console.log(1 + 5 % 3);
```

Question

What is the value of the following expression?

```
2 ** 2 ** 3 * 3
```

a. 768
b. 128
c. 12
d. 256

4.7. Other Operators

4.7.1. The String Operator `+`

So far we have only seen operators that work on operands which are of type `number`, but there are operators that work on other data types as well. In particular, the `+` operator can be used with `string` operands to **concatenate**, or join together two strings.

Example

`"Launch" + "Code"` evaluates to `"LaunchCode"`
Let's compare `+` used with numbers to `+` used with strings.

Example

```
 console.log(1 + 1);
1console.log("1" + "1");
2
```

Console Output

```
2
11
```

This example demonstrates that **the operator + behaves differently based on the data type of its operands.**

Warning

So far we have only seen examples of operators working with data of like type. For the examples `1 + 1` and `"1" + "1"`, both operands are of type `number` and `string`, respectively.

We will explore such "mixed" operations in a later chapter.

4.7.2. Compound Assignment Operators

A common programming task is to update the value of a variable in reference to itself.

Example

```
 let x = 1;
1x = x + 1;
2
3console.log(x);
4
```

Console Output

```
2
```

Line 2 may seem odd to you at first, since it uses the value of the variable `x` to update `x` itself. This technique is not only legal in JavaScript (and programming in general) but is quite common. It essentially says, "update `x` to be one more than its current value."

This action is so common, in fact, that it has a shorthand operator, `+=`. The following example has the same behavior as the one above.

Example

```
let x = 1;
x += 1;

console.log(x);

```

Console Output

```
2
```

The expression `x += 1` is shorthand for `x = x + 1`.

There is an entire family of such shorthand operators, known as **compound assignment operators**.

Compound Assignment Operators

Operator name	Shorthand	Meaning
Addition assignment	`a += b`	`a = a + b`
Subtraction assignment	`a -= b`	`a = a - b`
Multiplication assignment	`a *= b`	`a = a * b`
Division assignment	`a /= b`	`a = a / b`

4.8. Input with `readline-sync`

`console.log` works fine for printing static (unchanging) messages to the screen. If we wanted to print a phrase greeting a specific user, then `console.log("Hello, Dave.");` would be OK as long as Dave is the actual user.

What if we want to greet someone else? We could change the string inside the `()` to be `'Hello, Sarah'` or `'Hello, Elastigirl'` or any other name we need. However, this is inefficient. Also, what if we do not know the name of the user beforehand? We need to make our code more general and able to respond to different conditions.

It would be great if we could ask the user to enter a name, store that string in a variable, and then print a personalized greeting using `console.log`. Variables to the rescue!

4.8.1. Requesting Data

To personalize the greeting, we have to get **input** from the user. This involves displaying a **prompt** on the screen (e.g. “Please enter a number: “), and then waiting for the user to respond. Whatever information the user enters gets stored for later use.

As we saw earlier, each programming language has its own way of accomplishing the same task. For example, the Python syntax is `input("Please enter your name: ")`, while C# uses `Console.ReadLine();`.

JavaScript also has a built-in module for collecting data from the user, called `readline-sync`. Unfortunately, using this module requires more than a single line of code.

4.8.2. Syntax

Gathering input from the user requires the following setup:

```
1 const input = require('readline-sync');
2
3 let info = input.question("Question text... ");
```

There is a lot going on here behind the scenes, but for now you should follow this bit of wisdom:

I turn the key, and it goes.

Most of us do not need to know all the details about how cars, phones, or microwave ovens work. We just know enough to interact with them in our day to day lives. Similarly, we do not need to understand how `readline-sync` works at this time. We just need to know enough to collect information from a user.

As you move through the course, you WILL learn about all of the pieces that fit together to make this process work. For now, here is a brief overview.

4.8.2.1. Load the Module

In line 1, `const input = require('readline-sync')` pulls in all the functions that allow us to get data from the user and assigns them to the variable `input`.

Recall that `const` ensures that `input` cannot be changed.

4.8.2.2. How to Prompt the User

To display a prompt and wait for a response, we use the following syntax: `let info = input.question("Question text... ");`.

When JavaScript evaluates the expression, it follows the instructions:

1. Display `Question text` on the screen.
2. Wait for the user to respond.
3. Store the data in the variable `info`.

For our greeting program, we would code `let name = input.question("Enter your name: ");`. The user enters a name and presses the Return or Enter key. When this happens, any text entered is collected by the input function and stored in `name`.

Try It

Let's play around with the `input` statement. Open the repl.it link below and click the "Run" button.

```
1 const input = require('readline-sync');
2
3 let name = input.question("Enter your name: ");
```

repl.it

Note that after entering a name, the program does not actually DO anything with the information. If we want to print the data as part of a message, we need to put `name` inside a `console.log` statement.

After line 3, add `console.log("Hello, " + name + "!");`, then run the code several times, trying different responses to the input prompt.

By storing the user's name inside `name`, we gain the ability to hold onto the data and use it when and where we see fit.

Try adding another `+ name` term inside the `console.log` statement and see what happens. Next, add code to prompt the user for a second name. Store the response in `otherName`, then print both names using `console.log`.

Try It

Update your code to request a user's first and last name, then print an output that looks like:

```
First name: Elite
Last name: Coder
Last, First: Coder, Elite
```

4.8.3. Critical Input Detail

There is one very important quirk about the input function that we need to remember.
Given `console.log(7 + 2);`, the output would be `9`.

Now explore the following code, which prompts the user for two numbers and then prints their sum:

```
1 const input = require('readline-sync');
2
3 let num1 = input.question("Enter a number: ");
4 let num2 = input.question("Enter another number: ");
```

```
5
6console.log(num1 + num2);
```

repl.it

Run the program, enter your choice of numbers, and examine the output. Do you see what you expected?

If we enter `7` and `2`, we expect an output of `9`. We do NOT expect `72`, but that is the result printed. What gives?!?!?

The quirk with the `input` function is that it *treats all entries as strings*, so numbers get concatenated rather than added. Just like "Hello, " + "World" outputs as `Hello, World`, "7" + "2" outputs as `72`.

JavaScript treats input entries as strings!

If we want our program to perform math on the entered numbers, we must use type conversion to change the string values into numbers.

Try It

1. Use `Number` to convert `num1` and `num2` from strings to numbers. Run the program and examine the result.
2. Instead of using two steps to assign `num1` and then convert it, combine the steps in line 3.
   Place `input.question("Enter a number: ")` inside the `Number` function. Run the program and examine the result.
3. Repeat step 2 for `num2`
4. What happens if a user enters `Hi` instead of a number?

4.8.4. Check Your Understanding

Question

What is printed when the following program runs?

```
 const input = require('readline-sync');
1
2let info = input.question("Please enter your age: ");
3//The user enters 25.
4
5console.log(typeof info);
6
```

a. `string`
b. `number`
c. `info`
d. `25`

4.9. Exercises: Data and Variables

Exercises appear regularly in the book. Just like the concept checks, these exercises will check your understanding of the topics in this chapter. They also provide good practice for the new skills.

Unlike the concept checks, you will need a code editor to complete the exercises. Fortunately, you created a free account on Repl.it as part of the prep work.

Open this link to the repl.it file prepared for this task, then follow the instructions below.

4.9.1. Practice

Use the information given below about your space shuttle to complete the exercises:

Data	Value
Name of the space shuttle	Determination
Shuttle Speed (mph)	17,500
Distance to Mars (km)	225,000,000
Distance to the Moon (km)	384,400
Miles per kilometer	0.621

1. **Declare and assign variables**

   For each row in the table above, declare and assign variables.

   Hint: When declaring and assigning your variables, remember that you will need to use that variable throughout the rest of the exercises. Make sure that you are using the correct data type!

2. **Print out the type of each variable**

   For each variable you declared in the previous section, use the `typeof` operator to print its type to the console.

3. **Calculate a space mission!**

   We need to determine how many days it will take to reach Mars.

   a. Create and assign a miles to Mars variable. You can get the miles to Mars by multiplying the distance to Mars in kilometers by the miles per kilometer.
   b. Next, we need a variable to hold the hours it would take to get to Mars. To get the hours, you need to divide the distance to Mars in miles by the shuttle's speed.

c. Finally, declare a variable and assign it the value of days to Mars. In order to get the days it will take to reach Mars, you need to divide the hours it will take to reach Mars by 24.

4. **Print out the results of your calculations**

   Using the variable from part 3c above, print to the screen a sentence that says `"_____ will take ___ days to reach Mars."`

   Fill in the blanks with the shuttle name and the calculated time.

5. **Now calculate a trip to the Moon**

   Repeat the calculations, but this time determine the number of days it would take to travel to the Moon and print to the screen a sentence that says `"_____ will take ___ days to reach the Moon."`.

4.10. Studio: Data and Variables

In this studio, you are going to write code to display the *very important* **Launch Checklist LC04**.

LC04 displays information related to the space shuttle, astronauts, and rockets before launch.

4.10.1. Before You Start

If you are enrolled in a LaunchCode program, access this studio by following the repl.it classroom links posted in your class at learn.launchcode.org.

If you are working through this material on your own, use the repl.it links contained on this page.

4.10.2. Declare and Initialize Variables

Click this link to open the starter repl.it file.

Declare and initialize a variable for every data point listed in the table below. Remember to account for the different data types.

Note

For now, use the `string` type for the `date` and `time` values. Later in the class, we will learn other ways to work with date and time in JavaScript.

Variable	Value
`date`	Monday 2019-03-18
`time`	10:05:34 AM
`astronautCount`	7
`astronautStatus`	ready
`averageAstronautMassKg`	80.7
`crewMassKg`	`astronautCount` * `averageAstronautMassKg`
`fuelMassKg`	760,000

Variable	Value
`shuttleMassKg`	74842.31
`totalMassKg`	`crewMassKg` + `fuelMassKg` + `shuttleMassKg`
`fuelTempCelsius`	-225
`fuelLevel`	100%
`weatherStatus`	clear

4.10.3. Generate the LC04 Form

Display **LC04** to the console using the variables you declared and initialized.

The generated report should look *exactly* like the example below—including spaces and symbols (-, >, and *).

4.10.3.1. Example Output

Note that your output will should change when you assign different values to `astronautCount`, `fuelMassKg`, etc. The point is to AVOID coding specific values into the `console.log` statements. Use your variable names instead.

```
-------------------------------------
> LC04 - LAUNCH CHECKLIST
-------------------------------------
Date: Monday 2019-03-18
Time: 10:05:34 AM

-------------------------------------
> ASTRONAUT INFO
-------------------------------------
* count: 7
* status: ready

-------------------------------------
> FUEL DATA
-------------------------------------
* Fuel temp celsius: -225 C
* Fuel level: 100%

-------------------------------------
> WEIGHT DATA
-------------------------------------
* Crew mass: 564.9 kg
* Fuel mass: 760000 kg
* Shuttle mass: 74842.31 kg
```

```
* Total mass: 835407.21 kg

-----------------------------------
> FLIGHT PLAN
-----------------------------------
* weather: clear

-----------------------------------
> OVERALL STATUS
-----------------------------------
* Clear for takeoff: YES
```

4.10.4. Show Off Your Code

When finished, show your code to your TA so they can verify your work.

If you do not finish before the end of class, login to Repl.it and save your work. Complete the studio at home, then copy the URL for your project and submit it to your TA.

4.10.5. Bonus Mission

Use `readline-sync` to prompt the user to enter the value for `astronautCount`.

The values printed for `astronautCount`, `crewMassKg`, and `totalMassKg` should change based on the number of astronauts on the shuttle. (Don’t forget to convert the input value from a string to a number).