Contents

1 Mathematical Foundations 5
  1.1 Predicate Logic .............................................. 5
  1.2 Sets ................................................................ 6
  1.3 Bags, Lists, and Tuples ...................................... 7
  1.4 Graphs ............................................................... 7
  1.5 Summations and Recurrences ............................... 9

2 Web Technologies 11
  2.1 Content, Presentation, and Logic .......................... 11
  2.2 HTML ................................................................. 11
  2.3 CSS ................................................................ 12
  2.4 JavaScript .......................................................... 13
    2.4.1 Variable Declarations .................................... 14
    2.4.2 Implicit Conversions ...................................... 14
    2.4.3 Null and Undefined Values ............................. 15
    2.4.4 Interpolated Strings ...................................... 15
    2.4.5 Object Literals .............................................. 16
    2.4.6 Collections .................................................. 16
    2.4.7 For-Each Loops .......................................... 17
    2.4.8 Classes ......................................................... 17
    2.4.9 Anonymous Functions .................................. 18
Chapter 1

Mathematical Foundations

Software engineering means more than just writing code. In particular, an important part of the engineering side of the discipline is that software engineers must design and build software so that they are able to make justified guarantees about what that software does and does not do—they must be able to ensure that it meets its requirements. As in other areas of engineering, this is the math-heavy part of the discipline.

This chapter therefore provides a summary of the basic mathematical concepts that the following chapters build on. You should already be familiar with these concepts from earlier courses.

1.1 Predicate Logic

A boolean value is a value that is either false (written 0) or true (written 1). Boolean-valued functions are called predicates, and a predicate is said to hold on an input or be satisfied by an input when that input causes it to give the result true. For instance, the function $x \mapsto x < 0$ (i.e., the function whose result is the boolean $x < 0$ for any value $x$) is a predicate that holds on and is satisfied only by negative numbers.

Several operators can be applied to booleans. The unary negation operator ($\neg$, “not”) reverses its input, changing false to true and true to false, the binary conjunction operator ($\cdot \land \cdot$, “and”) has a true result only if both of its operands are true, and the binary disjunction operator ($\cdot \lor \cdot$, “or”) has a false result only if both of its operands are false. Another common binary operator is material implication ($\cdot \rightarrow \cdot$, “implies”) the boolean analog to “less than or equal to”. That is, the expression $P \rightarrow Q$ is true when $Q$ is at least as true as $P$, so that if $P$ is true, $Q$ must also be true.

These operators are summarized in the tables below:

<table>
<thead>
<tr>
<th></th>
<th>$0 \land 0 = 0$</th>
<th>$0 \lor 0 = 0$</th>
<th>$0 \rightarrow 0 = 1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\neg 0 = 1$</td>
<td>$0 \land 1 = 0$</td>
<td>$0 \lor 1 = 1$</td>
<td>$0 \rightarrow 1 = 1$</td>
</tr>
<tr>
<td>$\neg 1 = 0$</td>
<td>$1 \land 0 = 0$</td>
<td>$1 \lor 0 = 1$</td>
<td>$1 \rightarrow 0 = 0$</td>
</tr>
<tr>
<td></td>
<td>$1 \land 1 = 1$</td>
<td>$1 \lor 1 = 1$</td>
<td>$1 \rightarrow 1 = 1$</td>
</tr>
</tbody>
</table>

Negation has higher precedence than conjunction, followed by disjunction and then material implication. For example, the expression $P \rightarrow Q \lor \neg R \land S$ is equivalent to the parenthesized expression $P \rightarrow (Q \lor ((\neg R) \land S))$.

1 Another common convention is to write false as $\bot$ and true as $\top$. However, this convention tends to conflict with those symbols’ other possible meanings. For instance, this book uses $\bot$ to denote an undefined value.
A quantifier summarizes the evaluation of a boolean expression over a whole universe. The existential quantifier (∃, “there exists”) is used to write expressions of the form ∃x.φ(x), which are true if and only if there is a value for x that makes the subexpression φ(x) true. Likewise, expressions of the form ∀x.φ(x), using the universal quantifier (∀, “for all”), are true if and only if there is no value for x that makes φ(x) false. In both cases the expression is said to “quantify over” the variable x.

The order of a predicate logic determines to what extent quantifiers can appear in expressions. A zeroth-order predicate logic disallows quantifiers, whereas a first-order predicate logic allows quantifiers over variables with non-predicate, non-collection types, and higher-order logics allow even further quantification. First-order logic is generally sufficient for reasoning about whether software meets its requirements.

1.2 Sets

A set is a collection of values, called the set’s elements, without order or multiplicity. That is, no element of a set comes before or after any other, and while an element might be in or not in a set, it is not sensible to talk about the number of times an element is in a set.

A finite set can be written by extension, i.e., by listing its elements. Examples include ∅, the symbol for the empty set, which has no elements, and {0, 1, 2}, a set with the elements zero, one, and two. Sets can also be described by intension, by giving a predicate that holds on elements of the set and does not hold on any other values;² an expression describing a set this way is called a set comprehension. For instance, the comprehension \{x | x = 0 ∨ x = 1 ∨ x = 2\} gives the set corresponding to the predicate x → x = 0 ∨ x = 1 ∨ x = 2, which is just the set \{0, 1, 2\}.

The membership operator (· ∈ ·, “in” or “element of”) and its negation (· ∉ ·, “not in” or “not an element of”) test whether an element is in a set. For example, 2 ∈ \{0, 1, 2\} is true because two is an element of the set, but 4 ∈ \{0, 1, 2\} is false. The membership operator can also be used to write bounded quantifiers. Using a bounded existential quantifier, an expression like ∃x ∈ S.φ(x) is shorthand for the quantified conjunction ∃x.(x ∈ S) ∧ φ(x). However, using a bounded universal quantifier, an expression like ∀x ∈ S.φ(x) has implication as the unwritten operator: ∀x.(x ∈ S) → φ(x). Similar abbreviation of a conjunction is possible in a set comprehension; the comprehension \{x ∈ S | φ(x)\} is merely shorthand for \{x | (x ∈ S) ∧ φ(x)\}.

The containment operator (· ⊆ ·, “is contained by” or “is a subset of”) tests whether every element of one set is also in another. For instance, \{0, 2\} ⊆ \{0, 1, 2\} is true because both zero and two are elements of the set on the right, but \{0, 3\} ⊆ \{0, 1, 2\} is false because three is not. If S ⊆ T is true for two sets S and T, then S is called a subset of T, and T is called a superset of S.

Any set S can be wholly described by the predicate x → x ∈ S, the set’s indicator function, and so, for each boolean operator, there is a corresponding operation on sets. The unary complement operator (¬, “complement of”) produces a set with precisely the elements not in the original set, the binary intersection operator (· ∩ ·, “intersect” or “cap”) produces a set with the elements that the two original sets have in common, the binary union operator (· ∪ ·, “union” or “cup”) produces a set with the elements that are in either of the two original sets, and the set-difference operator (· \ ·, “without”) produces a set with the elements from the first operand that are not in the second. The relationships via indicator functions between these operators and the boolean operators are summarized at the top of the next page.

²A collection defined by a predicate is a class, and not all classes are sets, so in the general case, one must be careful to choose a predicate that will produce an actual set. However, classes that are not sets are extremely rare when reasoning about software, so the issue does not come up much in practice.
Two sets are called disjoint if they have no elements in common, or, equivalently, if their intersection is empty. A partition of a set is an arrangement of its elements into disjoint nonempty subsets called parts such that the union of all of the parts is the original set. A partition into two parts is called a bipartition, a partition into three parts is a tripartition, etc.

Finally, the cardinality operator (|·|, “cardinality of”) counts the number of elements in a set. So |{0, 1, 2}| = 3, and |{x | x < 0}| = ∞.\(^3\)

1.3 Bags, Lists, and Tuples

Other collections build on sets by tracking more information about their elements. The most basic is a bag, which is like a set, but allows repeated elements. For example, the set \{0, 0, 1, 2\} is the same as the set \{0, 1, 2\}, just written differently, but the bag \{0, 0, 1, 2\} is distinct from \{0, 1, 2\} because the former contains the element 0 twice.

A list is like a bag, but also tracks order. For instance, although the sets \{0, 1, 2\} and \{2, 0, 1\} are the same (again, just written differently), the lists \{0, 1, 2\} and \{2, 0, 1\} are distinct. Lists also support the indexing operator (\[·\]|·, “at index”) such that \(L[i]\) means the element of \(L\) at index \(i\), counting from zero,\(^4\) the concatenation operator (\(· ++ ·\), “concatenated with”) such that \(L ++ K\) means the list with the elements of \(L\) followed by the elements of \(K\), and lexicographic comparison (\(· < ·\), “lexicographically less than”) such that \(L < K\) is true if and only if, at the first index where \(L\) and \(K\) differ, \(L\) either has no element or has an element less than the corresponding one in \(K\).

A \(n\)-tuple is like a list, except that it has a specified and fixed length. For example, to track a point in 3D space, one might reasonably use a 3-tuple like \((x, y, z)\), but a list would be inappropriate because it does not make sense to allow fewer or more coordinates. \(n\)-tuples are sometimes referred to with natural-language words like “single” (when \(n = 1\)), “pair” (when \(n = 2\)), “triple” (when \(n = 3\)), etc.

Sets of \(n\)-tuples occur frequently as a generalization of functions; they are called relations, and a relation is said to be on a set if every element of its tuples is also an element of that set. A binary relation is a set of pairs, a ternary relation is a set of triples, etc. A binary relation is called symmetric if, for every pair \((x, y)\) in the relation, the pair \((y, x)\) is also present; otherwise the relation is asymmetric.

1.4 Graphs

A graph is a set of elements called vertices plus a binary relation on the vertices whose pairs are called edges. When defining a graph, one often writes an equation like \(G = (V, E)\) to quickly give variable names for the graph, its vertex set, and its edge set. When illustrating a graph, vertices are usually drawn as dots or circles, and edges are rendered as lines or arrows between vertices. When talking about a graph, an

---

\(^3\)Mathematicians are sometimes more specific, distinguishing between different kinds of infinity. These distinctions are rarely important when reasoning about software.

\(^4\)This convention is not universal; some authors use indices that count from one.
edge \((x, y)\) is usually described as “coming from” or “departing” \(x\) and “going to” or “arriving at” \(y\). The term *incidence* generalizes this concept; an edge \((x, y)\) is “incident on” both \(x\) and \(y\), but no other vertices. The *neighborhood* of a vertex is the set of other vertices that have a common incident edge; these other vertices are called *neighbors*.

A graph is simple if it has no self loop, that is, no edge from a vertex to itself. Graphs are assumed to be simple unless stated otherwise. The *complement* of a graph \(G\), written \(\overline{G}\), is a graph with the same vertex set but exactly the edges not appearing in \(G\), with one exception: when considering simple graphs, self loops are excluded from both \(G\) and \(\overline{G}\).

An undirected graph is a graph where the edge relation is symmetric, and any edge \((x, y)\) is thought of as being the same as its reversal \((y, x)\). A directed graph, on the other hand, does not guarantee such symmetry, so edges are usually considered distinct from their reversals.

A vertex-labeled graph \(G = (V, E, f)\) is a graph equipped with a function \(f\) that maps vertices to some other values. An edge-labeled graph is similar, except that the function maps edges to the values of interest. Two particular kinds of edge-labeled graphs are common: in a weighted graph, the edge labels are numbers called *weights* that in some sense represent the “lengths” of the edges, and an oriented graph is an undirected graph where each edge is labeled with an *orientation*, an ordering of its two vertices.

A subgraph of a graph \(G = (V, E)\) is a graph \(H\) whose vertex set and edge set are subsets of \(V\) and \(E\), respectively. This relationship is written \(H \subseteq G\). An induced subgraph is a subgraph with as many edges as possible given its vertices; the subgraph is said to be *induced by* its vertex set.

In an undirected graph \(G = (V, E)\), the degree of a vertex \(v\), written \(\deg(v)\), is the number of edges incident on it, or \(|\{u \mid (u, v) \in E\}|\). A vertex with degree zero is called independent, a vertex with degree one is called a pendant vertex (and its incident edge is called a pendant edge), and a vertex with the maximum possible degree is called universal. A set of vertices \(V'\) is called an independent set if every vertex in the subgraph induced on \(V'\) is independent; the set is called a clique if every vertex in the induced subgraph is universal. If an entire graph is an independent set, it is edgeless, and if an entire graph is a clique, it is called complete. The complete graph on \(n\) vertices is written \(K_n\). Furthermore, an \(n\)-partite graph is a graph whose vertex set can be partitioned into \(n\) independent sets. In particular, a bipartite graph can be partitioned into just two independent sets. The bipartite graph with parts of size \(x\) and \(y\) and as many edges as possible is written \(K_{x,y}\) where \(x\) is usually taken to be less than or equal to \(y\).

For a directed graph, degree is broken down into in-degree, the number of edges arriving at a vertex, which is \(|\{u \mid (u, v) \in E\}|\), and out-degree, the number of edges leaving a vertex, which is \(|\{w \mid (v, w) \in E\}|\). A vertex with in-degree zero is called a source, and vertex with out-degree zero is called a sink.

A walk is a sequence of vertices \([v_0, v_1, \ldots, v_n]\) and the edges \(((v_0, v_1), (v_1, v_2), \ldots, (v_{n-1}, v_n))\) between subsequent vertices. If the first and last vertices of a walk are different, the walk is said to be open; otherwise it is closed. A walk with no repeated edges is called a trail, and a walk with neither repeated edges nor repeated vertices—except perhaps a single repeated vertex to make the walk closed—is called simple.

Simple walks come up often enough to have special names. An open simple walk is called a path, and an undirected graph that is a path on \(n\) vertices is written \(P_n\). Similarly, a closed simple walk is called a cycle, and an undirected graph that is a cycle on \(n\) vertices is written \(C_n\).

A undirected graph is connected if any only if there is a path between every pair of distinct vertices. A connected maximal (this is, having as many vertices as possible) subgraph is called a component.

A undirected graph that contains no cycles is called a forest, and if the graph is also connected, it is called a tree. Within a forest or tree, a pendant vertex is called a leaf. A rooted tree is a tree where one vertex is designated as the tree’s root, and often all of the graph’s edges are directed or oriented towards or away from the root. When talking about a rooted tree, the term leaf usually refers to only pendant vertices that are not the root. A rooted forest is similar to a rooted tree, except that a root is designated for each component.
1.5 Summations and Recurrences

An expression of the form $\sum_{i=s}^{t} a_i$ is a definite summation, and is equivalent to the sum $a_s + a_{s+1} + \cdots + a_t$. For example, $\sum_{i=0}^{4} i + 2 = (0 + 2) + (1 + 2) + (2 + 2) + (3 + 2) + (4 + 2) = 20$. An expression of the form $\sum_{i=s}^{\infty} a_i$ is an indefinite summation, and is shorthand for $\lim_{t \to \infty} \sum_{i=s}^{t} a_i$.

Summation-like notation can also be used for other operations. For instance, the expression $\prod_{i=s}^{t} a_i$ stands for the product $a_s \cdot a_{s+1} \cdot \cdots \cdot a_t$, the expression $\bigwedge_{i=s}^{t} a_i$ stands for the conjunction $a_s \land a_{s+1} \land \cdots \land a_t$, the expression $\bigcup_{i=s}^{t} a_i$ stands for the union $a_s \cup a_{s+1} \cup \cdots \cup a_t$, and so on.

A recurrence is a definition of a function in terms of that same function applied to different arguments. For example,

$$f(n) = \begin{cases} f(n-1) + 2 & \text{if } n > 0 \\ 0 & \text{otherwise} \end{cases}$$

is a recurrence that defines the same function as the non-recurrence

$$f(n) = \begin{cases} 2n & \text{if } n > 0 \\ 0 & \text{otherwise} \end{cases}.$$ 

Expressions or equations that include summation-like subexpressions (even implicitly, by using ellipses) and/or recurrences are said to be open; expressions or equations without these features are called closed. Closing a summation or recurrence is the process of rewriting it as a closed expression.
Chapter 2

Web Technologies

With traditional software, a user must go through an installation process before being able to run a program, and subsequent upgrades also tend to involve user intervention. However, modern computer systems have several ways of supporting installation-free applications, programs that a user can run without any visible installation process. By far the most popular way to build an installation-free application is to write it as a web application, a system with a client-side component that is designed to be downloaded over a network and run in a web browser.

For various historical reasons, there are three languages that every modern browser provides support for: hypertext markup language (HTML), cascading style sheets (CSS), and JavaScript (more accurately referred to as ECMAScript). Therefore, while web applications’ server sides may be written in any language, their client sides typically use these three languages or at least languages that can be transpiled (automatically translated) to these three.

This chapter provides a brief overview of HTML, CSS, and JavaScript.

2.1 Content, Presentation, and Logic

As you have seen from working in other languages, modern GUI frameworks separate the code describing a GUI from the code describing its behavior; in this way cosmetic and behavioral changes can be kept separate. Unlike most GUIs, however, the GUIs for web applications are consumed in a variety of ways: users may interact with them via a desktop browser, a mobile browser, or a screenreader, and programs like webcrawlers or mashups may process the GUI without any human involvement at all. Consequently, web technologies not only separate GUI descriptions from logic, they also break GUI descriptions down into content (the information displayed along with annotations about its structure and meaning) and presentation (rules about how that content should be expressed under various circumstances).

2.2 HTML

HTML 5 is the de-facto content language of the modern web. While webpages have always used some version of HTML, until about 2016 the state of the language was more complicated: previous versions were plagued by inconsistent support across browsers, and web developers often had to use shims, or pieces of code that would compensate for missing or incorrectly implemented language features. The need for these workarounds is far less common today, but you may still find them in legacy code.
As a content language, HTML is roughly analogous to other content languages, such as Kv. The main difference is that elements (the HTML analog to widgets) in HTML 5 are chosen only for their semantics, not for their appearance or behavior. For example, HTML 5 has a nav element, which is used to group together content that makes up a page-navigation menu, but the presence of this element tells us nothing about how that menu is implemented—two different nav elements may look and behave completely differently.

The syntax for HTML 5 comprises several key forms:

- An HTML 5 doctype declaration, written <!doctype html>, must appear at the very beginning of a valid HTML 5 file. This doctype declaration distinguishes the file’s code from code written in similar languages like SGML or earlier versions of HTML.

- Void elements, or elements of types that can never contain other content, are written as an opening tag, or an element type name in angle brackets. For instance, the code <img> creates an element with the type img, i.e., an element for displaying an image. If you have experience working in similar languages, note that void elements in HTML 5 are never written with a trailing slash.

- Nonvoid elements, or elements of types that can contain other content, are written as an opening tag, followed by the content, followed by a closing tag, which is written the same way as an opening tag, but with a slash before the element type name. For example, the code <p>[example]</p> creates an element with the type p (a paragraph) and the text [example] as its content.

- Element attributes are the HTML equivalent to widget properties, though HTML 5 again restricts itself to semantic attributes—presentation properties are managed separately by CSS. An attribute is written as an attribute name followed by an equals sign and a double-quoted attribute value, all in the element’s opening tag. For instance, <img src="..."> is an image with a source URL, and <p class="foo" id="bar"></p> is an empty paragraph with the class foo and the ID bar.

- Entities represent characters, usually characters that would otherwise have a special meaning in HTML, like angle brackets or quotation marks. An entity is written as an ampersand followed by an entity code and a semicolon. For example, &lt; is a less-than sign, and &amp; is an ampersand.

For historical reasons, HTML is a do-what-I-mean (DWIM) language; if a developer writes invalid HTML, the browser will attempt to guess what they meant instead of reporting an error. This can be problematic from a software engineering perspective because it makes bugs and potential bugs harder to notice. Therefore, it is common to use a tool called a validator to check HTML code for errors before deployment.

### 2.3 CSS

CSS, and in particular CSS 3, is the de-facto presentation language of the modern web. As with HTML, good cross-browser support for CSS is a relatively recent phenomenon; shims are common in legacy code.

CSS, like HTML, is not a general purpose programming language. Instead, a CSS file is a collection of rules about how HTML elements should be presented; these rules are organized into rulesets. Each ruleset has two parts: (1) a selector that determines when the ruleset applies to an element and (2) a collection of declarations, each of which is a property name paired with an associated value.

Whenever a browser needs to determine the value of a presentation property for an element, it (effectively) follows a three-step process: it catalogs all of the available CSS rulesets whose selectors match the element in question (including any rulesets built into the browser), it sorts these rulesets according to the specificity of their selectors, and finally it locates the most specific mention of the property in question. Thus, a developer

\[1\] The older parts of HTML often use abbreviations that seem unnecessary from a modern perspective.
can make sweeping presentational changes by writing a very general CSS ruleset, but still override particular elements’ properties by introducing other rulesets with more specific selectors.

When using this rule-based approach, developers often need to organize elements into groups that can be selected all at once; such groups are called classes. Support for classes is built in to HTML; it provides not only an id attribute as in other GUI frameworks, but also a class attribute, which is a space-separated list of the names of the classes to which an element belongs.

The syntax for CSS is very loosely based on the syntax for the C language. For example, the following is a CSS ruleset that formats attributions in block quotations:

```css
blockquote cite.with-dash {
    display: block;
    margin-top: -0.5em;
    text-align: right;
}
```

The part before the left curly brace is the selector, which is made up of three basic selectors: `blockquote`, `cite`, and `.with-dash`. Each basic selector has two parts: an optional prefix character (which determines what kind of selection is made) and a value to be matched. For instance, the prefix character in `.with-dash` is a dot, which means the selector considers elements’ classes, so `.with-dash` matches all elements in the class `with-dash`. On the other hand, `blockquote` has no prefix character, which means that it considers elements’ types and therefore matches all `blockquote` elements, i.e., all block quotations.

Selectors can also be combined using various operators. For example, `cite.with-dash` combines the basic selectors `cite` and `.with-dash` using the conjunction operator (which is written as no text at all) to create a new selector that matches only elements that are both attributions and in the `with-dash` class. Likewise, `blockquote cite.with-dash` combines `blockquote` and `cite.with-dash` using the “within” operator (written as a space) to create a selector that matches only `with-dash` attributions that are inside block quotations.

Between the curly braces are three declarations, each of which ends with a semicolon. Otherwise their syntax is very similar to the syntax for properties in Kv. For instance, `display: block;` is a declaration that sets these attributions’ `display` property to the value `block`, making each attribution appear on a line by itself.

Like HTML, CSS is a DWIM language. Again, common practice is to use a validator to check CSS code before deployment.

### 2.4 JavaScript

Finally, JavaScript is a general-purpose programming language that is most commonly used for web applications, both on the client side (running in a browser) and on the server side (running in an environment like Node.js), but that has also been adopted for other purposes, such as mobile-app development (using frameworks like Angular).

Historically, JavaScript was designed for programming in the small. At the time it was first created, the expectation was that most interactive webpages would be written using Java applets, so JavaScript was meant to play the role of “Java’s little brother”, being suitable for smaller programs, perhaps around ten lines long. But Java applets never became widely popular, and developers began writing JavaScript programs with rather more than ten lines. JavaScript, as a consequence, was forced to play catch-up, incorporating features to support software engineering practices that were never relevant to its original purpose while still trying to maintain backwards compatibility. It eventually became recognizable as a modern language around version 6, after heavy borrowing from Python.
JavaScript, like Python, is implicitly duck typed, and although it uses prototype-based inheritance (a non-standard approach for supporting object-oriented programming), the class syntax introduced in version 6 makes that distinction immaterial for most applications. Still, JavaScript does differ from Python in two main ways. First, its syntax was designed to imitate Java, so JavaScript uses curly braces and semicolons instead of colons and indentation. Second, JavaScript is a DWIM language, so validation before deployment is important.

Below follows a sampling of the features and quirks to be aware of when working in JavaScript.

### 2.4.1 Variable Declarations

In modern JavaScript, variables can be declared in two ways. A variable declared with the `let` keyword can be assigned to normally:

```javascript
let variable = 0;
variable = 1; // no error
```

whereas, as protection against unintended modification, a variable declared with `const` can be assigned to only in its declaration:

```javascript
const variable = 0;
variable = 1; // TypeError: invalid assignment to const
```

Note that this meaning of `const` in JavaScript is different from its meaning in languages like C++, where something that is `const` cannot be changed at all. In JavaScript, an object in a `const` variable is allowed to have its fields changed:

```javascript
const anObject = new Object();
anObject.field = 1; // no error
```

because it is only assignments to the variable that are forbidden.

For backwards compatibility, it is still possible, though frowned upon, to declare variables with the keyword `var`:

```javascript
var variable; // old-style declaration (discouraged)
variable = 1;
```

Declarations with `var` behave almost like declarations with `let`, except that they are *hoisted*, i.e., effectively moved earlier in the code, which can lead to surprises:

```javascript
variable = 1; // confusingly assigns to a variable declared below
var variable; // old-style declaration (discouraged)
```

Also for backwards compatibility, it is possible to assign to a variable without declaring it:

```javascript
variable = 1; // without declaration (discouraged)
```

but then the variable automatically becomes global, so functions can accidentally overwrite each other’s data by happening to use the same variable name.

### 2.4.2 Implicit Conversions

As a DWIM language, JavaScript has operators that do not complain about mismatched types, but handle mismatches by applying automatic conversions. For instance, in JavaScript the comparison `false == '0'`
is true because both values are automatically converted to the number zero, but while \texttt{false \texttt{==} '00'} is also true for similar reasons, the expression \texttt{'0' \texttt{==} '00'} is not true because both values are strings, so no automatic conversion occurs. The possibility of automatic conversion has proven bug-prone, especially in the case of equality comparisons, so JavaScript now has the \textit{strict comparison operators} \texttt{===} and \texttt{!==}, which check equality without coercing values to different types. Modern JavaScript code uses these operators almost exclusively; the \texttt{==} and \texttt{!=} operators are almost never the better choice.

Another quirk due to these automatic conversions is that logical operators can be applied in surprising ways to non-boolean values. For example, JavaScript implements the \texttt{||} operator as “if the first argument is \textit{truthy} (a value that automatically converts to true), return it; otherwise, if the first argument is \textit{falsey} (it automatically converts to false), return the second argument”, which some developers exploit to write code like:

\begin{verbatim}
// change variable to defaultValue if variable is falsey
variable = variable || defaultValue;
\end{verbatim}

Such code is terse, but not especially readable to programmers coming from other languages. It can also be fault-prone, because, for example, the developer might intend \texttt{defaultValue} to be used only when \texttt{variable} is undefined, but forget that other values like 0 and '' also count as falsey.

### 2.4.3 Null and Undefined Values

When JavaScript was originally designed, it borrowed some ideas from Java, like the use of \texttt{null} to represent “no object”, but not other ideas, like signaling errors by throwing exceptions. These two choices eventually led to a difficult question about what should happen when code attempts to access a nonexistent field. On one hand, the code could not throw an exception, so it would have to produce some value, and \texttt{null} would seem to be the only reasonable choice. On the other hand, then the program would not be able to notice the problem if the field it was trying to access could legitimately be null. To escape the dilemma, the designers chose to create another special value, \texttt{undefined}, specifically for this situation:

\begin{verbatim}
const anObject = new Object();
const variable = anObject.field; // no error; sets variable to undefined
\end{verbatim}

Once exceptions were added to JavaScript, there was no longer a motivation for \texttt{undefined}, but developers were already using it for other purposes, and it had to be kept for backwards compatibility.

Thus, where Java has just \texttt{null}, and Python has just \texttt{None}, JavaScript has two ways to indicate “no value”: \texttt{null} and \texttt{undefined}. While many developers have strong opinions, there is no universal consensus on when to use each, so to avoid bugs when maintaining JavaScript code, it is important to learn what convention the project is following.

### 2.4.4 Interpolated Strings

Like many other scripting languages, JavaScript has \textit{interpolated strings}, concatenations that are written as strings with embedded code; in JavaScript these strings are called \textit{template literals}. The process of evaluating such strings is called \textit{interpolation}, a tongue-in-cheek allusion to the fact that it fills in gaps in the template literals with data.
JavaScript template literals are written with backquotes, and placeholders in template literals are written as expressions surrounded by curly braces and prefixed by a dollar sign. For instance, the following code sets `message` to the string "The answer is 5."

```javascript
const augend = 3;
const addend = 2;
const message = 'The answer is ${ augend + addend }.';
```

Template literals are generally preferred over concatenation when some of the values being concatenated are fixed text, and others are not.

### 2.4.5 Object Literals

While JavaScript does support custom classes, as described in Section 2.4.8, it also provides a shorthand syntax for creating prepopulated objects of type `Object`. This syntax is identical to the Python syntax for a dictionary with string keys, except that the quotes around field names may be omitted when those field names are valid identifiers. For example, the following code creates an `Object` that describes a measurement:

```javascript
const measurement = { value: 2.0, units: 'kg' };
```

Such object literals have at least three common uses. First, for data types without methods, especially those that are only used in one or two places, it may not be worth the effort and maintenance cost to write a dedicated class. Object literals provide an alternative. Second, configuration settings are typically written as object literals because the syntax makes it easy to leave out fields that the user wants to be filled in with default values. Third, object literals see frequent use for data that needs to be serialized because JavaScript has built-in support for such serialization.

You may encounter another use in legacy code: code written before JavaScript had a proper dictionary data type would often abuse objects as dictionaries, storing key/value pairs as the names and values of fields. This kludge, however, limited keys to strings and made it tedious to implement dictionary-processing correctly, because it would have to account for, for example, ignoring any fields and methods inherited from the `Object` class. Thus, modern code tends to use JavaScript's `Map` type instead.

### 2.4.6 Collections

JavaScript has a list type called `Array`, a set type called `Set`, and dictionary type called `Map`. It does not have immutable equivalents; the convention for JavaScript is to always use mutable collections.

JavaScript arrays are written like Python lists:

```javascript
const list = [0, 1, 2];
```

but they behave more like the `List` type in Java than the `list` type in Python. The one major exception is that JavaScript arrays have no bounds checking, so attempts to read from out-of-bounds indices just result in `undefined`:

```javascript
const list = [0, 1, 2];
const element = list[-1]; // no error; sets element to undefined
```

and attempts to write to out-of-bounds indices might expand the array:

```javascript
const list = [0, 1, 2];
list[9] = 3; // no error; expands list to be ten elements long
```

This makes out-of-bounds errors somewhat more difficult to detect and debug in JavaScript.
Lists can also be used in assignments the way tuples are in Python, even in declarations:

```javascript
// set first to 1 and second to 2
const [first, second] = [1, 2];
```

In JavaScript terminology, such assignments are called *destructuring assignments*.

JavaScript sets are created using `new`, and the constructor can optionally take another collection:

```javascript
const emptySet = new Set();
const nonemptySet = new Set([0, 1, 2]);
```

Likewise for JavaScript maps, though in that case the collection given to the constructor must be a collection of key/value pairs:

```javascript
const emptyMap = new Map();
const nonemptyMap = new Map([[0, 1], [2, 3]]);
```

Except in the case of the `[ ]` operator for accessing list elements, JavaScript collections are generally manipulated using methods, not operators. For instance, a map lookup uses the `get` method, not the `[ ]` operator:

```javascript
const map = new Map([[ 'a', 1], [ 'b', 2]]);
const value = map.get('a'); // correct; sets value to 1
const broken = map[ 'a' ]; // incorrect; actually tries to access map.a
```

### 2.4.7 For-Each Loops

JavaScript has for-each loops, which are written with the keywords `for` and `of`:

```javascript
const set = new Set([0, 1, 2]);
for (const element of set) {
  // process element
}
```

The keyword `of` should not be confused with `in`, which makes the loop iterate over an object’s fields’ names:

```javascript
const measurement = { value: 2.0, units: 'kg' };
for (const fieldName in measurement) {
  // process fieldName
  // (it might be 'value', or 'units', or the name of something inherited)
}
```

### 2.4.8 Classes

Classes in JavaScript are written using Java-like syntax, but without visibility modifiers:

```javascript
class SuperClass {
  // ...
}

class SubClass extends SuperClass {
  // ...
}
```
Methods are also written using Java-like syntax, but minus the types and visibility modifiers. Constructors must be named `constructor`, and methods that should not be called from outside the class are conventionally named with a leading underscore:

```javascript
class Example {
  constructor (parameter) {
    // ...
  }

  _protectedMethod (parameter, anotherParameter) {
    // ...
  }

  publicMethod () {
    // ...
  }
}
```

As in Python, JavaScript forbids method overloading, so each of a class’s methods must have a unique name, and, in particular, a class may not have multiple constructors.

Field accesses from within a method must always use this `this` keyword, and JavaScript getters and setters are written using the `get` and `set` keywords. For instance:

```javascript
class Example {
  constructor () {
    this._property = 0;
  }

  get property () {
    return this._property;
  }

  set property (value) {
    this._property = value;
  }
}
```

Also, like in Python, these getters and setters are called using the same syntax that one uses to access a field:

```javascript
const example = new Example();
example.property = 0;
const variable = example.property;
```

### 2.4.9 Anonymous Functions

JavaScript functions can be stored in variables and then called using those variables:

```javascript
const variable = function example () {
  // ...
};

variable();
```
In such cases it is possible to write an anonymous function, a function without a name:

```javascript
const variable = function() {
    // ...
};

variable();
```

Very similar to an anonymous function is an arrow function, known as a lambda in other languages. In the most general case, arrow functions are written as their parameters followed by an arrow, followed by their body. For example:

```javascript
const variable = (parameter, anotherParameter) => {
    return parameter + anotherParameter;
};

const result = variable(0, 1);
```

But arrow functions that consist of just a return statement can have their body shortened to just the expression to return:

```javascript
const variable = (parameter, anotherParameter) => parameter + anotherParameter;

const result = variable(0, 1);
```

And arrow functions with exactly one parameter may have the parentheses around that parameter dropped.

The differences between regular functions (including anonymous functions) and arrow functions have to do with variable capturing, the ability of a function to refer to variables declared outside of itself. Arrow functions are the simplest, as they capture all variables, so code like:

```javascript
function outer() {
    let counter = 0;
    const increment = () => {
        ++counter;
        return counter;
    };    
    return increment();
}

const result = outer();
```

will do what one might expect, incrementing the counter from zero to one and returning the value one to be stored in `result`.

But `increment` is not just a function, but a closure, meaning that it keeps these variables in existence as long as the inner function might be called. Therefore, this code:

```javascript
function outer() {
    let counter = 0;
    return () => {
        ++counter;
        return counter;
    }
}
```
const increment = outer();
let result = increment();
result = increment();

creates a counter and a function to increment it and returns the incrementing function. That incrementing function closes over the variable counter, keeping counter in existence. So the first time increment is called, counter increases to one, and the second time counter increases to two.

Regular functions also close over most variables, but there are a few exceptions, the most notable of which is the variable this. Usually there is no reason not to capture these variables, so inner functions are typically arrow functions, except in older code, which may have been written before arrow functions existed.