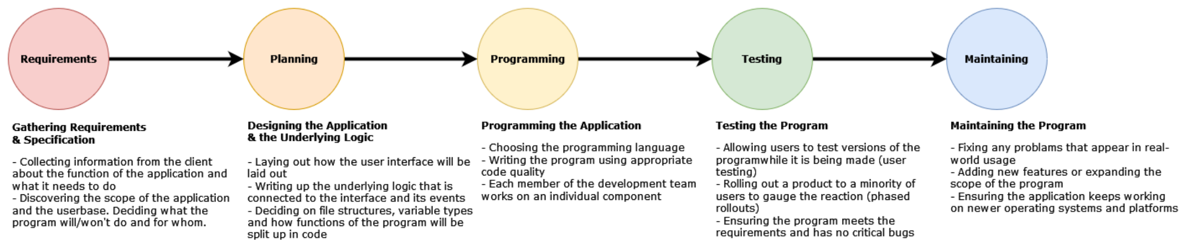
Unit 6, Assignment 2 – Design Workshop

Thomas Robinson

SOLIHULL COLLEGE

Thomas Robinson

Task One – Software Lifecycle & Structures (P5)



Gathering Requirements & Specification

- Collecting information from the client about the function of the application and what it needs to do
- Discovering the scope of the application and the userbase. Deciding what the program will/won't do and for whom.

Designing the Application & the Underlying Logic

- Laying out how the user interface will be laid out
- Writing up the underlying logic that is connected to the interface and its events
- Deciding on file structures, variable types and how functions of the program will be split up in code

Programming the Application

- Choosing the programming language
- Writing the program using appropriate code quality
- Each member of the development team works on an individual component

Testing the Program

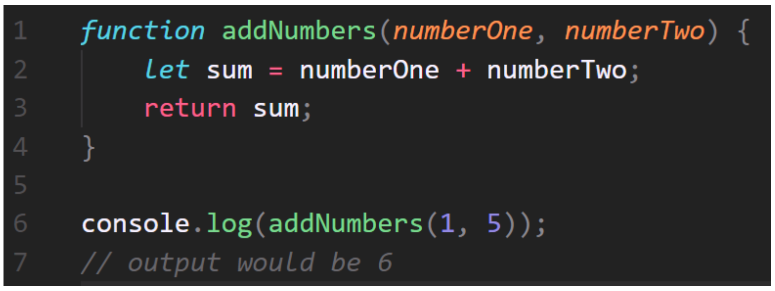
- Allowing users to test versions of the programwhile it is being made (user testing)
- Rolling out a product to a minority of users to gauge the reaction (phased rollouts)
- Ensuring the program meets the requirements and has no critical bugs

Maintaining the Program

- Fixing any problams that appear in real-world usage
- Adding new features or expanding the scope of the program
- Ensuring the application keeps working on newer operating systems and platforms

("Task 1 - Software Lifecycle.drawio.png")

Functions



```javascript
function addNumbers(numberOne, numberTwo) {
    let sum = numberOne + numberTwo;
    return sum;
}

console.log(addNumbers(1, 5));
// output would be 6
```

A piece of code that can be called upon many times by other parts of the program

Languages come with many built-in functions for tasks such as text manipulation, array modification, mathematical tasks and etc.

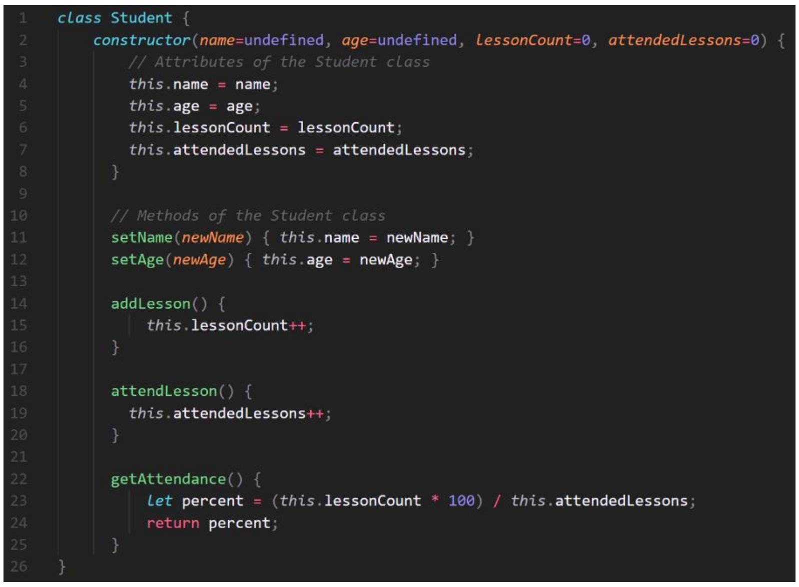
Procedures

Similar to a function but does not return a value.

Classes

Where an object type is defined in a program, see objects.

Objects



```javascript
class Student {
    constructor(name=undefined, age=undefined, lessonCount=0, attendedLessons=0) {
        // Attributes of the Student class
        this.name = name;
        this.age = age;
        this.lessonCount = lessonCount;
        this.attendedLessons = attendedLessons;
    }

    // Methods of the Student class
    setName(newName) { this.name = newName; }
    setAge(newAge) { this.age = newAge; }

    addLesson() {
        this.lessonCount++;
    }

    attendLesson() {
        this.attendedLessons++;
    }

    getAttendance() {
        let percent = (this.lessonCount * 100) / this.attendedLessons;
        return percent;
    }
}
```

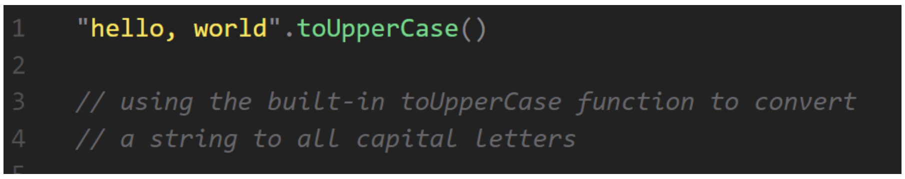
Represents an entity that can contain its own various properties and attributes (information about the object) and methods (functions that can be run against the object).

Data Abstraction

Only focusing on the necessary aspects of an object and filtering out those that may not be required or visible to a certain party.

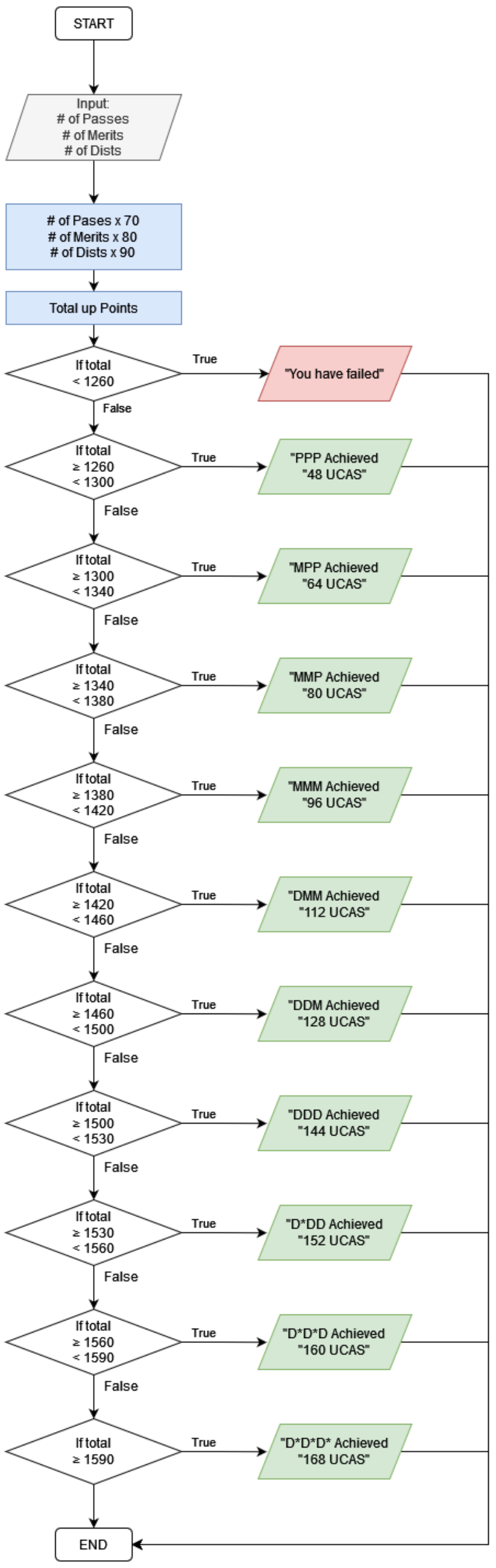
Pre-defined Code

Built-in variables and functions of a programming language and are already defined; their names cannot be reused as a variable name or a function name.



```
"hello, world".toUpperCase()

// using the built-in toUpperCase function to convert
// a string to all capital letters
```

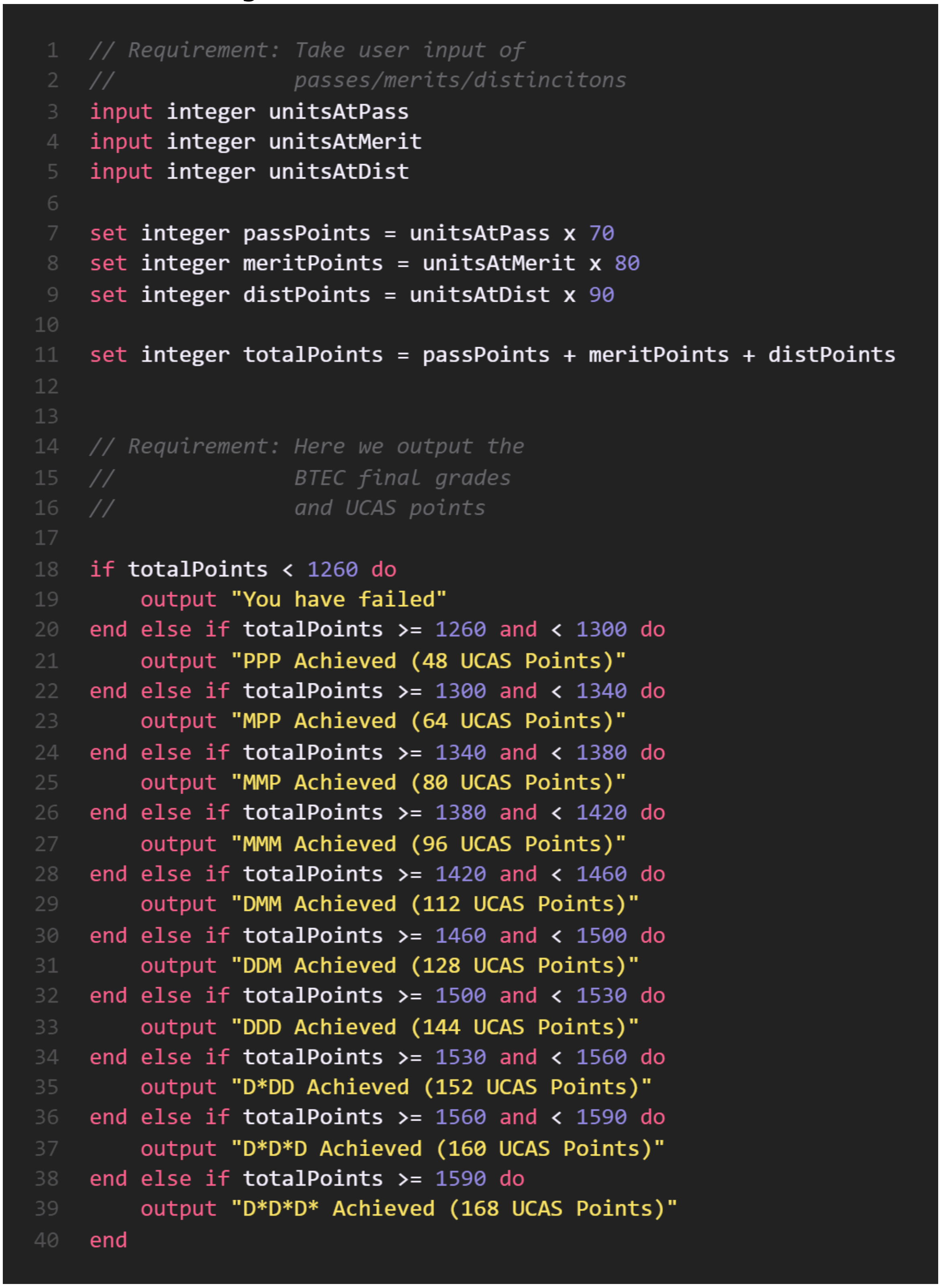
Task Two – Flowchart (P6)



("Task 2 - Flowchart.drawio.png")

Task Three – Algorithm (M2, D2)

Pseudocode Program



```
// Requirement: Take user input of
//              passes/merits/distincitons
input integer unitsAtPass
input integer unitsAtMerit
input integer unitsAtDist

set integer passPoints = unitsAtPass x 70
set integer meritPoints = unitsAtMerit x 80
set integer distPoints = unitsAtDist x 90

set integer totalPoints = passPoints + meritPoints + distPoints


// Requirement: Here we output the
//              BTEC final grades
//              and UCAS points

if totalPoints < 1260 do
    output "You have failed"
end else if totalPoints >= 1260 and < 1300 do
    output "PPP Achieved (48 UCAS Points)"
end else if totalPoints >= 1300 and < 1340 do
    output "MPP Achieved (64 UCAS Points)"
end else if totalPoints >= 1340 and < 1380 do
    output "MMP Achieved (80 UCAS Points)"
end else if totalPoints >= 1380 and < 1420 do
    output "MMM Achieved (96 UCAS Points)"
end else if totalPoints >= 1420 and < 1460 do
    output "DMM Achieved (112 UCAS Points)"
end else if totalPoints >= 1460 and < 1500 do
    output "DDM Achieved (128 UCAS Points)"
end else if totalPoints >= 1500 and < 1530 do
    output "DDD Achieved (144 UCAS Points)"
end else if totalPoints >= 1530 and < 1560 do
    output "D*DD Achieved (152 UCAS Points)"
end else if totalPoints >= 1560 and < 1590 do
    output "D*D*D Achieved (160 UCAS Points)"
end else if totalPoints >= 1590 do
    output "D*D*D* Achieved (168 UCAS Points)"
end
```

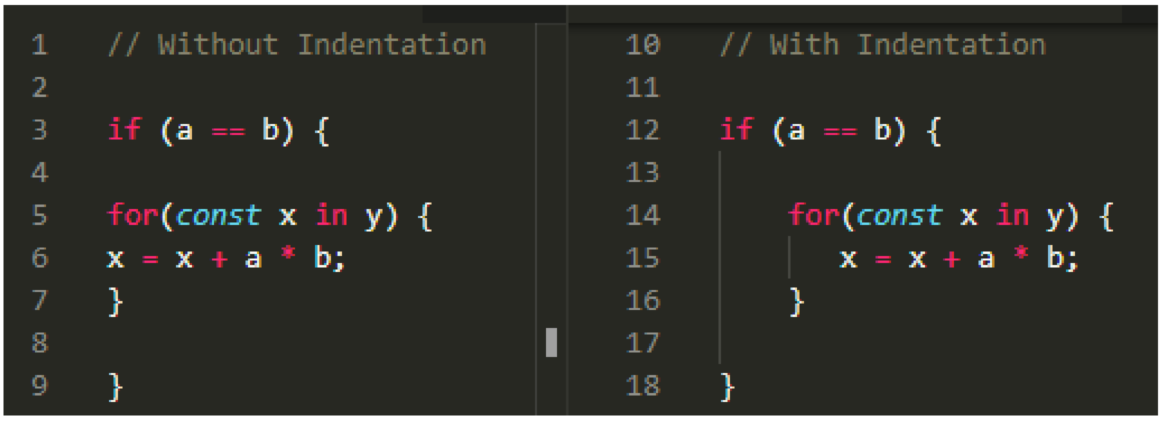
("Task 3 - Pseudocode.pseudo")

Data Types

Type/Struct	Variable Name	Reason for Choice
Int	unitsAtPass	User input of the number of units obtained at a pass grade. A whole number, used in later calculations.
Int	unitsAtMerit	User input of the number of units obtained at a merit grade. A whole number, used in later calculations.
int	unitsAtDist	User input of the number of units obtained at a distinction grade A whole number, used in later calculations.
Int	passPoints	The number of points obtained by units at a pass grade. A whole number, used in later calculations.
Int	meritPoints	The number of points obtained by units at a merit grade. A whole number, used in later calculations.
Int	distPoints	The number of points obtained by units at a merit grade. A whole number, used in later calculations.
Int	totalPoints	Used to store the total number of points. A whole number, used in later calculations.
Process		Calculate the number of points obtained by units at each grade by multiplying `unitsAt(Pass/Merit/Dist) * 70/80/90` then set the results to the `pass/merit/dist)Points` variables
Process		Calculate the total number of points by summing `(pass/merit/dist)Points`
If-else Statements		Used to calculate which message to show to the user (their expected grade and their UCAS points).

Task Four – Code Readability (D1)

Indentation



```
// Without Indentation

if (a == b) {

for(const x in y) {
x = x + a * b;
}

}
```

```
// With Indentation

if (a == b) {

    for(const x in y) {
        x = x + a * b;
    }

}
```

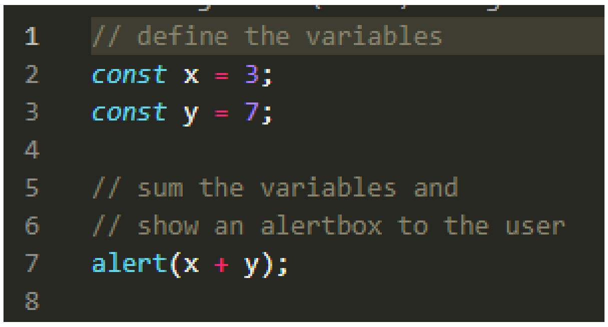
Indenting code is the process of adding spaces—or more commonly, tabs—to the start of lines of code to make it more clear which parts relate to which 'block'. For example, the code within an if-statement would be indented to make it clearer to understand.

Some languages, such as Python, rely on indentation and if the source code is not indented properly or consistently, the interpreter will be unable to run the understand the code.

In the example, the for loop is nested inside of the if statement and is indented. The content of the for loop is also indented.

Comments

Comments are parts of code that are not read by an interpreter or compiler. They allow developers to add descriptions or information about parts of code. This can help developers who are unfamiliar with the codebase to understand what each section is doing.



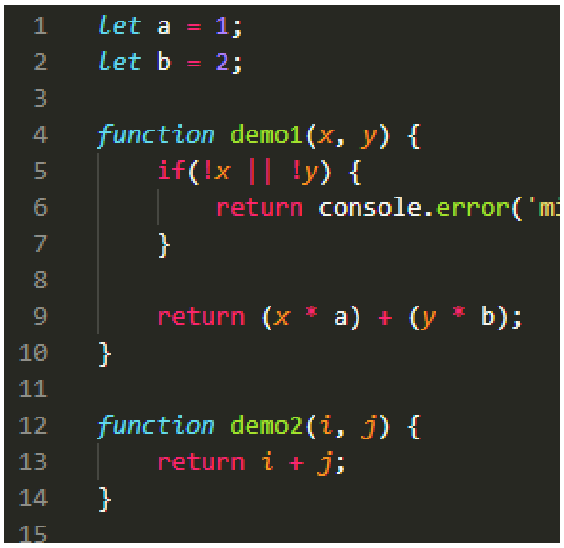
```
// define the variables
const x = 3;
const y = 7;

// sum the variables and
// show an alertbox to the user
alert(x + y);

```

In many languages, the way to create a comment is by prefixing a line with two forward slashes (//) or a hash (#). It may also be possible to create a multi-line comment, for example in CSS a comment is opened with "/*" and closed with "*/"

White Space

White space refers to blank lines. These can be used to space out and group certain lines of code, for example to separate initial variable declaration to functions underneath. White space also includes the space left by indentations and other empty areas.

In the below example, there is white space on lines 3, 8 and 11. These break up different areas in the code, making it easier to read.



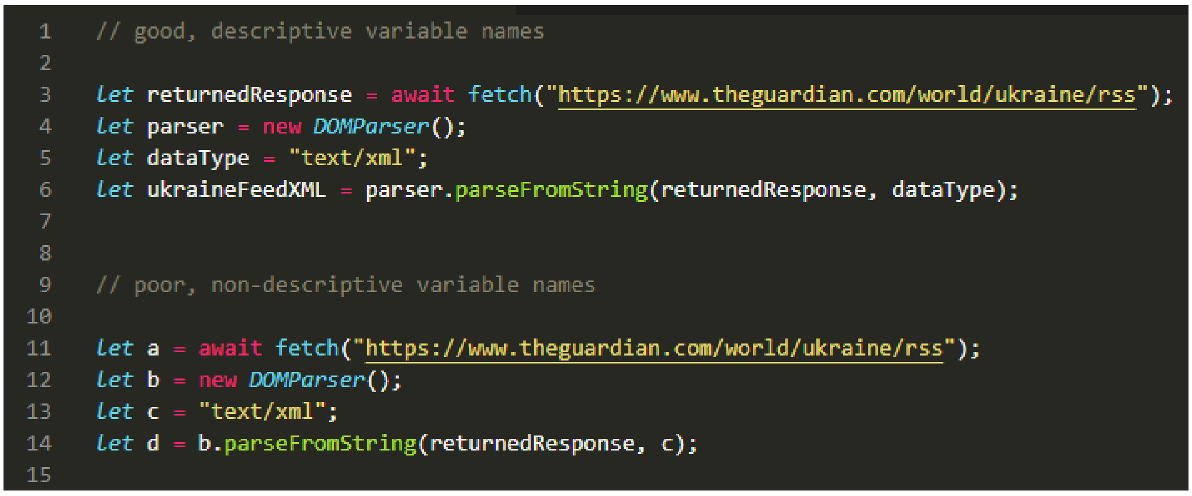
```
1   let a = 1;
2   let b = 2;
3
4   function demo1(x, y) {
5       if(!x || !y) {
6           return console.error('mi
7       }
8
9       return (x * a) + (y * b);
10  }
11
12  function demo2(i, j) {
13      return i + j;
14  }
15
```

Variable Names

Using accurate, descriptive variable names prevents a developer from getting confused and overwhelmed by information when programming. For example, naming a variable that contains the price of an item "item_price" rather than "x" ensures that there is no ambiguity as to what the variable should contain.

There are many ways of writing multi-word variable names. The most common are:

- Camel Case: helloWorld
- Pascal Case: HelloWorld
- Snake Case: hello_world
- Kebab Case: hello-world



```
// good, descriptive variable names

let returnedResponse = await fetch("https://www.theguardian.com/world/ukraine/rss");
let parser = new DOMParser();
let dataType = "text/xml";
let ukraineFeedXML = parser.parseFromString(returnedResponse, dataType);


// poor, non-descriptive variable names

let a = await fetch("https://www.theguardian.com/world/ukraine/rss");
let b = new DOMParser();
let c = "text/xml";
let d = b.parseFromString(returnedResponse, c);

```