Making Patterns with Code

DECO1012 - Design Programming

Week 2

THE UNIVERSITY OF SYDNEY

Recap

THE UNIVERSITY OF SYDNEY

Hello p5.js and Hello world!

- Introduced the basics of p5.js, a JavaScript library
- used the **print()** function to say "Hello world!" to the console
- Anatomy of a sketch: setup(), draw(), functions and function calls:

```
function setup()
{
    print("hello world!");
}
```

```
function draw()
{
    // code here
}
```

THE UNIVERSITY OF SYDNEY

Coordinates (x, y), line() & ellipse()

- p5.js uses x and y coordinates to place shapes on the canvas
- 0,0 is the top left of the canvas
- **line(x1, y1, x2, y2);** & **ellipse(x,y,w,[h]);**
- Built in variables
  - **windowWidth** & **windowHeight** = set the size of the canvas to window size
  - **width** & **height** = the width and height we set our canvas to
  - **mouseX** & **mouseY** = current coordinates of our mouse
- Order of execution
- Styling: fill(R, G, B), stroke(R, G, B), strokeWeight(), noFill(), noStroke();
- Operators: mouseX + 10, width/2

p5.js showcase

THE UNIVERSITY OF SYDNEY

https://editor.p5js.org/ishan5ain/sketches/rUCrLN9fD

The Basics: Variables

THE UNIVERSITY OF SYDNEY

Variables: Defining custom variables

Variables are what we use when we want to store information.

Defining a variable will look like something like this:

```
let myVar;
```

The **keyword** `let` is used to tell the computer that a variable named `myVar` exists.

`myVar` is not a great name for a variable because it's not very descriptive. It's fine for this example, but when you name your variables try to be descriptive.

THE UNIVERSITY OF SYDNEY

Naming Conventions

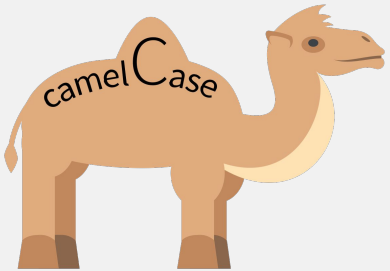
When naming variables and functions `myVar` is not the same as `myvar`. JavaScript is **case sensitive**.

You may have noticed that all the built-in variables and functions start with lowercase letters. This is a good convention to follow. The convention used is called **camelCase**.

Function and variable names can only contain letters, numbers, underscores, or dollar signs. They can not contain spaces.

Function and variable names can not start with a number.

There is a list of reserved words that `JavaScript` that cannot be used for a variable name. It is also possible to replace variables and function of `p5.js`, by using the same name. This is almost never a good practice.



Variables: Assigning

Assigning a value to a variable will look something like this:

```
myVar = 5;
```

The value `5` is now stored in the variable `myVar`

It is possible to define and assign a variable at the same time, like this:

```
let myVar = 5;
```

THE UNIVERSITY OF SYDNEY

Variables: Usage

Variables can be used to set other variables.

```
let bar = 5;

let foo = bar;

//foo now contains 5
```

Variables can be used as parameters in function calls.

```
let myVar = “Hello World”;

print(myVar);

//this will output ‘Hello World’ to console
```

Strings can be enclosed by ‘single quotation marks’, “double quotation marks” and `backticks`

Variables: Data Types

"JavaScript is a loosely typed and dynamic language. Variables in JavaScript are not directly associated with any particular value type, and any variable can be assigned (and re-assigned) values of all types"[1]

```
let myVar = 5;

myVar = 7.9;

myVar = true;

myVar = 'Hello World';
```

JavaScript has several data types.
These are the most common examples:
Number
Boolean
String

THE UNIVERSITY OF SYDNEY

[1] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures

Variables: Making use of Operators

It can be very useful to change the value of a variable based on its previous value.

```
let myVar = 4;

myVar = 3 * myVar;

//myVar now contains 12
```

```
//The addition operator also works with strings to concatenate them

let myVar = `Hello`;

myVar = myVar + “World”;

//myVar now contains ‘HelloWorld’
```

Shorthand Assignment Operators

There are also shorthand assignment operators that can be used:

`myVar += 5;` is the same as `myVar = myVar + 5;`

Shorthand assignment operators:

`+=  -=  *=  /=`

https://www.primogif.com/p/C9FZuXRrkXIY0

Increment and Decrement Operators

Postfix will increment or decrement after the expression is evaluated.

`A++` Postfix increment operator.

`A--` Postfix decrement operator.

```
myVar++;
```

is functionally the same as

```
myVar += 1;
```

and

```
myVar = myVar + 1;
```

Note: Prefix will increment or decrement before the expression is evaluated. But, this is rarely used.

`++A` Prefix increment operator.

`--A` Prefix decrement operator.

The Basics: Scope

Scope

Scope is a term we use to describe where variables and functions can be accessed.

The two scopes we are looking at in this lesson are:

1. Global Scope: Accessible **everywhere**
2. Block Scope: Only accessible between a pair of curly brackets { }

Scope: Global

Variables defined outside of functions, such as `setup()` and `draw()`, are accessible everywhere in your sketch and are considered to be within the **global scope**. This can feel convenient but it is recommended to use local variables unless a globally scoped variable is required.

Example:

```
let canvasW = 400;
let canvasH = 400;

function setup() {
  createCanvas(canvasW, canvasH);
}
```

THE UNIVERSITY OF SYDNEY

Scope: Block

Code blocks are denoted by the curly brackets {} and any variables defined **inside that code block** are only accessible inside that code block. This scope is similar to the function scope in that the variables are disposed of when the block has completed all of its instructions.

Variables defined using the let keyword are always confined to **code block scope**.

You may also see code using the const keyword. This is similar to let but the value cannot be reassigned. The keyword const isn't something you need to know or use yet.

Another keyword that allows you to define a variable is var. The difference between them is that var ignores the code block it is defined in and assumes **Prototype scope** *(Which isn't something we need to cover yet)*. We avoid using var in this course for simplicity.

Scope: Why not make everything Global?

You may be wondering why we don't simply define all variables in the **global scope,** here are a few good reasons:

- **Accidental modification of values:** If a function unintentionally uses the same variable name then it can access and overwrite the value in the global scope. If other functions depend on that value then this will result in errors and issues in the code.
- **Resource management:** For your browser to display web pages efficiently it needs to manage resources. This means releasing memory when its no longer used. Variables in the global scope won't get released because the browser thinks they are still needed.

THE UNIVERSITY OF SYDNEY

Scope: When to use Globals

- If you want to access the variable in both the setup() and draw() functions:
  - Variables defined inside a **block** are only available in that **scope**.
  - E.g. If you define a variable in the body of `setup()`, it will not be available inside `draw()`.
- If you want to retain the value between `draw()` calls:
  - Variables in **block scope** are disposed of after the block (e.g. a function) is exited.
  - E.g. If you define a variable inside `draw()` it will be disposed of and created fresh the next time `draw()` runs. You will not be able to keep the value between `draw()` calls.

THE UNIVERSITY OF SYDNEY

Global variables and `p5.js` functions/variables

The inbuilt variables and functions of `p5.js` can only be used in the `setup()` and `draw()` blocks of your sketch. This means that you cannot use those variables and functions to assign a value in the **global scope**, because `p5.js` functions aren't available there.

However, remember that you don't have to assign to a variable when you define it. Sometimes the solution for sharing values is to define the variable in **global scope** and then assign it later inside `setup()`. This might be useful if you need to share the variable, but can't call a function in the global scope of the sketch.

THE UNIVERSITY OF SYDNEY

Global variables and p5.js functions/variables

We briefly covered the data types that JavaScript makes available, but there are also some p5.js data types. One of those types is p5.Color and it is very useful for storing colours as it is accepted by most p5.js functions.

In order to assign this type to a variable you can use the p5 function color().

Example:

```
let myColor;

function setup() {
  createCanvas(400, 400);
  myColor = color(255, 0, 0);
  background(myColor);
}
```

Notice that color() and p5.Color are spelled using American English.

THE UNIVERSITY OF SYDNEY

The Basics: Conditionals

THE UNIVERSITY OF SYDNEY

The Boolean Data Type

Booleans are a JavaScript data type that can have two possible values: `true` and `false`.

Boolean values can be stored in variables. You can negate a boolean using an exclamation mark `!`.

```
let somethingTrue = true;
//somethingTrue now contains the value
true

let alsoTrue = !false;
//alsoTrue now contains the value true

let notTrue = !somethingTrue;
//notTrue now contains the value false
```

THE UNIVERSITY OF SYDNEY

Negation

This table lists the result of negating a boolean value. The left hand column lists possible values and the right hand column list the result of negating the value on the left.

p	!p
`true`	`false`
`false`	`true`

Booleans are not Strings

It is important to recognise that booleans are different to the string data type.

Strings are defined with backticks `example` , quotation marks ‘example’ and double-quotation marks “example”.

Whereas `true` and `false` are keywords.

‘true’ and `true` are not the same!

You can use the keyword `typeof` to check the data type:

```
typeof “false”;      //will return ‘string’
typeof false;        //will return ‘boolean’
```

THE UNIVERSITY OF SYDNEY

Relational Operators

Relational operators return a boolean based on the relational condition between two values.

`<`	Less than operator.
`>`	Greater than operator.
`<=`	Less than or equal operator.
`>=`	Greater than or equal operator.

Examples:

```
5 < 2; //evaluates as false
5 > 2; //evaluates as true
6 <= 5; //evaluates as false
6 >= 5; //evaluates as true
6 >= 6; //evaluates as true
```

All operators available in JavaScript are listed here:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators

THE UNIVERSITY OF SYDNEY

if() Statements

"The if statement executes a statement if a specified condition is truthy."[1]

Syntax

```
if (condition) {
   statement
}
```

Example

```
let myVar = 2;
if (myVar < 5) {
   print("myVar is less than 5");
}
```

[1] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/if...else

Equality Operators

Equality operators are expressed in a similar way to relational operators, but they instead evaluate for equality or inequality.

They also return a boolean based on two values.

`==` Equality operator.

`!=` Inequality operator.

Examples:

```
5 == 2; //evaluates as false
5 != 2; //evaluates as true
```

THE UNIVERSITY OF SYDNEY

Logical Operators

Logical operators are used to determine if multiple conditions are satisfied.

&& Logical AND.

|| Logical OR.

It is important to note that relational and equality operators are evaluated independently and so they are **not** associative.

Incorrect:
```
if(a < b < c){

}
```

Should be

Correct:
```
if(a < b && b < c){

}
```

THE UNIVERSITY OF SYDNEY

Logical Operators: logic table

This table lists the result of the || (or) and && (and) logical operators on a pair of booleans.

p	q	p || q	p && q
true	true	true	true
false	true	true	false
true	false	true	false
false	false	false	false

else

A useful keyword to use with an if statement is else. It allows you to execute code based on either boolean result of the condition. The else clause (block) is run if the conditional of the if returns false.

Syntax

```
if (condition) {
    statement1 //runs if condition is true
} else {
    statement2 //runs if condition is false
}
```

else if()

Conditional statements can be chained with `else if`.

Syntax

```
if(condition1) {
    statement1 //runs if condition1 is true
} else if(condition2){
    Statement2 //runs if condition1 is false and condition2 is true
} else if(condition3){
    statement3 //runs if all previous conditions are false and condition3 is true
} else if(conditionN){
    statementN //runs if all previous conditions are false and conditionN is true
}
```

Nesting

Code blocks can be nested to adjust your codes behaviour. Nesting `if` statements allows you to create branching decisions.

Take note of the indentations →. We use Tab to identify nested code blocks.

This convention greatly improves readability.

```
if(animal == "cat"){
    //the animal is a cat
    if(owner == "me"){
        //i am the owner of this cat
        if(!catFedToday){
            //the cat has not been fed today
            feedCat();
            catFedToday = true;
        } else {
            //the cat has been fed
            patCat();
        }
    } else if(owner == "neighbour"){
        //this cat belongs to my neighbour
        patCat();
    }
}
```

Note: The indentation arrows have been made visible in this example for illustration purposes. In the p5 editor they will appear as white space.

The Basics: Loops

THE UNIVERSITY OF SYDNEY

for Statements

When we want to run similar code multiple times, we can use a `for` loop.

Condition
This condition is checked at the **start** of each iteration. If it evaluates to `true`, the loop will run.

```
for(let i = 0; i < 10; i++){
    print(i);
}
```

Initialisation
Here we initialise variables for use in the for loop.

You are not limited to using only this variable within the statement, any variable within scope is available.

Final Expression
At the **end** of each iteration the final expression will be evaluated.

Statement
This is the code we want to run at each iteration.

The curly brackets are optional if you only want to run one statement, however it is a good convention to include them for readability.

THE UNIVERSITY OF SYDNEY

Nested for loops

`for` loops can be nested just like other code blocks. This is very handy for generating grids.

Example:

```
for (let x = 0; x < 3; x++) {
  for (let y = 0; y < 4; y++) {
     print("x==" + x + " y==" + y);
  }
}
```

Output:

```
x==0 y==0
x==0 y==1
x==0 y==2
x==0 y==3
x==1 y==0
x==1 y==1
x==1 y==2
x==1 y==3
x==2 y==0
x==2 y==1
x==2 y==2
x==2 y==3
```

Handy hint: Strings can be concatenated with other variables to produce longer string. You can see this being use in the example above in the statement `print("x==" + x + " y==" + y);`

Common issue: Infinite Loops

Something to be **avoided** is infinite loops. This can occur when the condition of a `for` statement is always `true`.

Example: This code will initialise `i` as 1 and then increment `i` by 1 every loop. So the condition '`i` is greater than `0`' will always evaluate as `true`.

```
for(let i = 1; i > 0; i++){

    print(i);

}
```

An infinite loop can produce issues with your code that can be hard to track down because they **will not** report an error in your console. An indication that you have an infinite loop in your code is that your output won't update and your browser may freeze or run slowly.

THE UNIVERSITY OF SYDNEY

Nested for loops

`for` loops can be nested just like other code blocks. This is very handy for generating grids.

Example:

```
for (let x = 0; x < 3; x++) {
  for (let y = 0; y < 4; y++) {
     print("x==" + x + " y==" + y);
  }
}
```

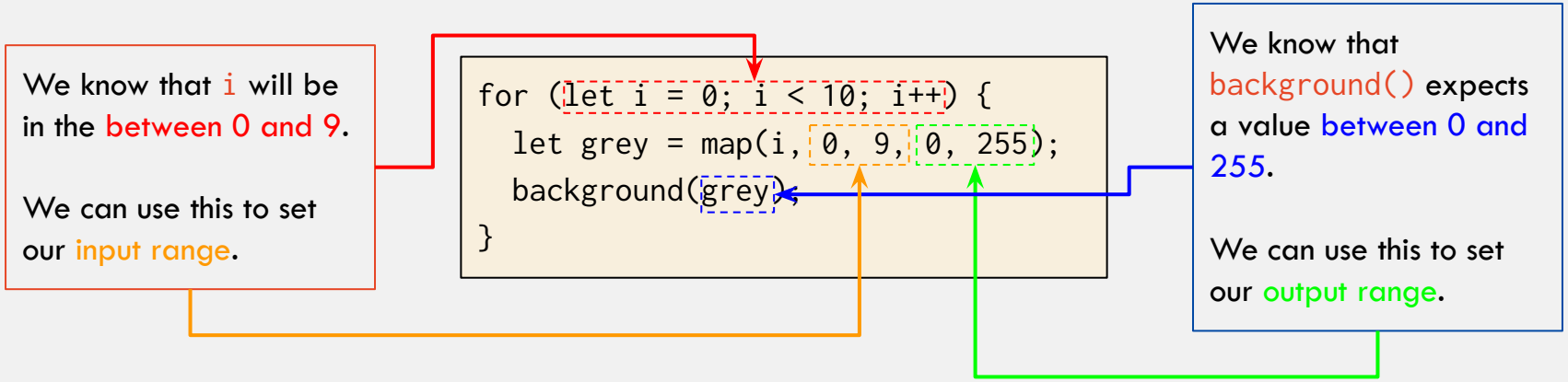
Output:

```
x==0 y==0
x==0 y==1
x==0 y==2
x==0 y==3
x==1 y==0
x==1 y==1
x==1 y==2
x==1 y==3
x==2 y==0
x==2 y==1
x==2 y==2
x==2 y==3
```

Handy hint: Strings can be concatenated with other variables to produce longer string. You can see this being use in the example above in the statement `print("x==" + x + " y==" + y);`

map() function

`map()` is great to pair with `for` loops because it can remap iterators to a new scale.

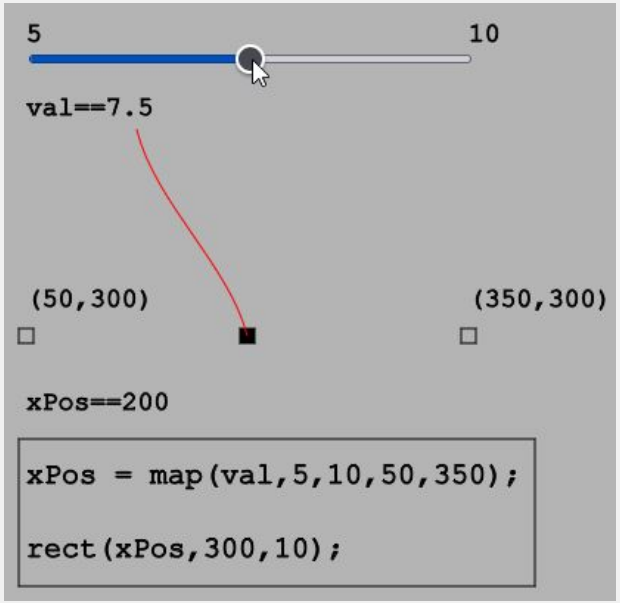


THE UNIVERSITY OF SYDNEY

Other great uses are positioning elements with relative values derived from `width` and `height`.

map() function

A map() is a useful function that remaps a value within a range to a new range. This is a very powerful tool when used to create relative values.



The Remainder Operator %

The remainder operator % is an arithmetic operator which returns the remainder left over when one value is divided by another. This provides us with an excellent way to repeatedly cycle through a range of numbers.

This is often used to pick odd and even numbers, can you work out how to do this?

Example:

```
for (let i = 0; i <= 6; i++) {
  print(i + " % 3 == " + (i % 3));
}
```

Output:

```
0 % 3 == 0
1 % 3 == 1
2 % 3 == 2
3 % 3 == 0
4 % 3 == 1
5 % 3 == 2
6 % 3 == 0
```

THE UNIVERSITY OF SYDNEY

The Remainder Operator %

The remainder operator % returns a negative number when the first value (on the left hand side) is negative.

Example:

```
for (let i = 0; i >= -7; i--) {
  print(i + " % 4 == " + (i % 4));
}
```

Output:

```
 0 % 4 ==  0
-1 % 4 == -1
-2 % 4 == -2
-3 % 4 == -3
-4 % 4 ==  0
-5 % 4 == -1
-6 % 4 == -2
-7 % 4 == -3
```

THE UNIVERSITY OF SYDNEY

Before we wrap up…

Remember to practice coding throughout the week. Can you write a sketch to classify odd and even numbers?

Try out simple versions of the code we've talked about today and look ahead at this week's challenges before the your tutorial. Practice helps and this way you can come ready with questions!

Questions? Post your coding questions on the canvas discussion boards.

If you need to talk to your tutor, arrange for a meeting using Canvas (option to setup a meeting available in the Home page).

THE UNIVERSITY OF SYDNEY

See you next week!

Design Programming - Week 2 Lecture

THE UNIVERSITY OF SYDNEY