COMP3311 Week7 – Introduction to PHP

Syntax

Variables

Functions

Arrays

Conditions

Loops

Syntax

Syntax is the set of rules about how code is interpreted

PHP syntax is somewhat similar to Java and Perl

Always end PHP statements with a semicolon `;`

Comments start with `//` or `#` for a line comment

Start and end with `/*` and `*/` for multi-line comments

Variables

Variables don't need to be declared before use.

The following *declares* a variable (named $name) and *assigns* it a value at the same time:

```
$name = "Leonardo";
```

Here, `$name` is the variable, `"Leonardo"` is a string literal, and = is the assignment operator.

There is no need to declare the **type** of the variable

An aside about string literals

PHP string literals come in a few forms.
The two most common are:

`'Single-quoted strings'`, and

`"Double-quoted strings"`

The difference is that double-quoted strings do *variable interpolation.* That is, you can embed variables in double-quoted strings.

```
echo "My favourite goat is $name";
```

Variables - types

PHP is *dynamically typed*.
Variables can store whatever you like, and aren't stuck storing the same type for their whole lives.

This is a bad idea, but valid and correct PHP:

```
$x = 10; // store an integer
$x = "No thanks"; // store a string
$x = array(1,2,3); // store an array
echo $x;
```

Variable names

A valid variable name starts with a letter or underscore, followed by any number of letters, digits, or underscores

Variable names are **case sensitive**.

A few styles:

"Camel case": `$userName;`

"Pascal case": `$UserName;`

"Snake case": `$user_name;`

"Camel Snake": `$User_Name;`

Functions

At their core, **functions** are used for organizing code to avoid repetitive work and to simplify code

Same idea as **methods** in Java, **subroutines and functions** in Visual Basic, and **functions** in C

The PHP library and extensions supply a very large number of useful functions (over 1400 on PHP 5.6)

Arguments and return values

Functions:

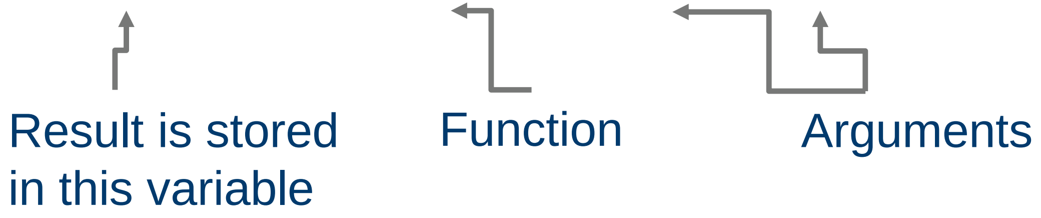
- Take zero or more *arguments* or *parameters*.
  These control the behaviour of the function
- Return an **optional** *result* or *return value.*
  Multiple return values can be simulated by returning an array instead
  You don't need to use the result

Example of a built-in function

The following piece of code calls a built-in function called `substr`.

`substr` takes a string and one or two more arguments, and returns a single result

```
$name = "Leonardo";
$last_bit = substr($name, 3);
```



Create your own functions

Create functions using the **function** keyword

```
function do_something_amazing($x, $y) {
    return $x * $y;
}

echo do_something_amazing(2, 5);
```

The arguments are called **$x** and **$y** when in the function

Argument evaluation

Arguments are **evaluated** before being passed to the function

```
function do_something_amazing($x, $y) {
    return $x * $y;
}

echo do_something_amazing(2 + 1, 5 - 4);
```

$x will be **3** and **$y** will be **1** when in the function

Overloading & default arguments

Overloading: you can't declare multiple functions with the same name but different arguments in PHP

You can work around this by using different function names, or using *default arguments*

```
function do_something_amazing($x, $y = 3) {
    return $x * $y;
}

echo do_something_amazing(4);     // = 12
echo do_something_amazing(4, 4);  // = 16
```

Passing arguments by reference

PHP has an *unusual* idea of how references work.
We will avoid them as much as possible

You can declare that a function takes a *reference to a variable* as an argument using **&**

This lets the function change the contents of a variable passed in.

```
function swap (&$a, &$b) {
   $temp = $a;
   $a = $b;        // assigning to $a and $b
   $b = $temp; // changes their values outside
}
```

A problem

Consider the following code:

```
$s1 = "John";
$s2 = "Dorothy";
$s3 = "Sasha";
$s4 = "Deanna";
print "Students are $s1, $s2, $s3, $s4";
```

What if we want to add more students?

What if we had a few thousand?

Arrays to the rescue

The Array data type lets you store lots of values in a single variable.

```
// create an array
$students = array("John", "Dorothy", "Sasha", "Deanna");

// Add (or replace) students in the array
$students[4] = "Wendy";

// Append by not including the index
$students[] = "Alice";

// look up index 1
print $students[1];
```

"New" array syntax

PHP 5.4 introduced a shorter array syntax:

```
$students = ["John", "Dorothy", "Sasha", "Deanna"];
```

Feel free to use it, but example code will use `array()`

Alternative ways to create arrays

Alternatively, pretend your array exists and just start jamming stuff into it.

```
// Create and populate an array.
$students[0] = "John";
$students[1] = "Dorothy";
```

This works using the append syntax too

```
// Create and populate an array
$students[] = "John";
$students[] = "Dorothy";
```

Types of PHP array

PHP arrays can be used in two ways:

- Sequential arrays
- Associative arrays

Excitingly, both are declared using the same syntax, and you can even mix and match.

Sequential arrays are what we've just seen - an in-order list of stuff that you can look up by index

Indexes are zero-based - the first element is at [0]

Associative arrays

Associative arrays map a **key** to a **value**.
These are like a Map or HashMap in Java, except ordered

```
$nicknames = array(
    'Leonardo' => 'Leo',
    'Donatello' => 'Donnie',
    'Michelangelo' => 'Mikey',
    'Raphael' => 'Raph',
);

// Look up by key
print $nicknames['Leonardo'];
```

What can you store in arrays?

Numbers

```
$a = array(1, 2, 3);
```

Strings

```
$a = array('a', 'b', 'c');
```

Other arrays

```
$a = array(
    'numbers' => array(1, 2, 3),
    'letters' => array('a', 'b', 'c')
)
```

Mixtures of types

```
$a = array(1, 'b', 3, array('what?'));
```

Arrays inside arrays?

Let's have another look at that array of arrays

```
$a = array(
    'numbers' => array(1, 2, 3),
    'letters' => array('a', 'b', 'c')
);
```

How do we use this?

How can we find the value at index 1 in the numbers array?

Accessing nested arrays

Look up by key or index with brackets [ ]

```
$a = array(
    'numbers' => array(1, 2, 3),
    'letters' => array('a', 'b', 'c')
);

$numbers_array = $a['numbers'];
$index_one = $numbers_array[1];

echo $index_one; // prints 2
```

Ditch the intermediaries

`$numbers_array` is the same as `$a['numbers']`

We can substitute one for the other, algebra style
(not always possible, but in this simple case it holds up)

```
$numbers_array = $a['numbers'];
$index_one = $numbers_array[1];

// can be replaced by
$index_one = $a['numbers'][1];
```

This works for arbitrarily deep nesting

Conditionals

These work the same way as you'd expect from Java or similar languages, using the *if* statement.

```
$a = 1 + 9;
if($a === 10) {
    echo "MATHS RULES";
} else {
    echo "Nothing makes sense";
}
```

The whole *else* block is optional

Comparison

Note this from the last slide:

```
if($a === 10)
```

There are a number of *comparison operators*.

`===` is the *identical* comparison operator
`==` is the *equal* comparison operator

Identical requires both sides to be the **same type**.
Equal will do some tricks called *type juggling* so it can compare them.
Mostly it tries to **convert arguments to numbers**.

Stick with using `===` unless you are have good reason

Examples of comparison chaos

From the PHP documentation:

```
echo (0 == "a");          // 0 == 0 -> true
echo ("1" == "01");       // 1 == 1 -> true
echo ("10" == "1e1");     // 10 == 10 -> true
echo (100 == "1e2");      // 100 == 100 -> true
```

It can be even worse:

```
echo (1 == "1 million"); // 1 == 1 -> true

echo (1 === "1 million"); // false, better!
```

Switch statement

Again, very similar to Java, C, C++ and related languages

Use caution: *switch* uses equality operator rules!

```
switch ($a) {
    case 0:
    case 1:
        echo '$a is zero or one';
        break;
    case 2:
        echo '$a is two';
        break;
    default:
        echo '$a is something else';
}
```

Without `break;` control "falls through"

For loops

For loops follow this general structure:

```
for (initializer; loop condition; increment) {
  loop body
}
```

- *initializer* runs before the loop
- *loop condition* gets tested before every loop
- *loop body* gets executed every loop
- *increment* happens at the end of every loop

Any or all of these can be empty

Classic for loop example

A simple counter

```
for ($i=0; $i < 10; $i++) {
    // remember, we're generating HTML
    print "<p>$i</p>";
}
```

- `$i=0` runs before the loop
- `$i < 10` gets tested before every loop
- `$i++` happens at the end of every loop

While loops

While loops are a bit simpler
They only have a loop condition

```
while (loop condition) {
  loop body
}
```

loop condition gets tested before every loop
loop body gets executed every loop

Only the loop body can be empty

Convert *for* loop to *while* loop

The *for* loop from before can be rewritten as a while loop

```
$i=0; // initializer
while($i < 10) { // loop condition
    // remember, we're generating HTML
    print "<p>$i</p>";

    $i++; // increment
}
```

Prefer *for* loops for this sort of code, it's easy to forget the increment in a while loop

Using a for loop on an array

You can use the **count** function to get the length of an array

```
for ($i=0; $i < count($my_array); $i++) {
    // The . operator concatenates strings
    print "<p>" . $my_array[$i] . "</p>";
}
```

This only works for *dense* arrays, not *sparse* arrays.

```
$my_array[1000] = 'hello'; // this will break
```

Better loops through arrays - *foreach*

The *foreach* loop is specifically designed for looping through arrays, both sequential and associative

```
$students = array("John", "Dorothy", "Sasha", "Deanna");

foreach($students as $student) {
    echo "<p>" . $student . "</p>";
}
```

This works on sparse arrays too

foreach with associative arrays

Applying the same foreach loop to associative arrays loops through the *values*, not the keys

```
$nicknames = array(
    'Leonardo' => 'Leo',
    'Donatello' => 'Donnie',
    'Michelangelo' => 'Mikey',
    'Raphael' => 'Raph',
);

foreach($nicknames as $nick) {
    echo "<p>$nick</p>"; // Leo Donnie Mikey Raph
}
```

foreach for associative array keys

Use **=>** in the foreach loop to get access to the keys when looping

```
$nicknames = array(
    'Leonardo' => 'Leo',
    'Donatello' => 'Donnie',
    'Michelangelo' => 'Mikey',
    'Raphael' => 'Raph',
);

foreach($nicknames as $fn => $nick) {
    echo "<p>$nick is $fn</p>"; // Leo is Leonardo, etc.
}
```

Putting it all together

```php
function starts_with($long_string, $short_string) {
    $short_length = strlen($short_string);
    $long_truncated = substr($long_string, 0, $short_length);
    return $long_truncated === $short_string;
}
function find_truncations($array) {
    foreach($array as $key => $value) {
        if(starts_with($key, $value)) {
            echo "<p>$key starts with $value";
        }
    }
}
$nicknames = array(
    'Leonardo' => 'Leo', 'Donatello' => 'Donnie',
    'Michaelangelo' => 'Mikey', 'Raphael' => 'Raph',
);
find_truncations($nicknames);
```

Work your way through this code slowly, try to understand what each part does.
Where are the functions called?
Are there any loops or conditionals?

Output of Previous Code

```
Leonardo starts with Leo

Raphael starts with Raph
```

Use, for instance, http://phptester.net/ to see the above output