Chapter 4

Expressions and Control Flow in PHP

True or False

```php
<?php
    $x = 3;
    $y = 3 * (abs(2 * $x) + 4);
    echo "a: [" . $y > 10 . "]<br>";
    echo "b: [" . $y == 10 . "]<br>";
?>
```

Output:

```
a: [1]
b: []
```

```php
<?php
echo "a: [" . TRUE . "]<br>";
echo "b: [" . FALSE . "]<br>";
?>
```

Output:

a: [1]
b: []

- PHP arbitrarily assigns a numerical value of 1 to TRUE. However, the constant FALSE is defined as NULL, another predefined constant that denotes nothing.

PHP Operators

PHP divides the operators in the following groups:

Operator	Description	Example
Arithmetic	Basic mathematics	`$a + $b`
Array	Array union	`$a + $b`
Assignment	Assign values	`$a = $b + 23`
Bitwise	Manipulate bits within bytes	`12 ^ 9`

PHP Operators

PHP divides the operators in the following groups:

Operator	Description	Example
Comparison	Compare two values	`$a < $b`
Execution	Execute contents of backticks	`` `ls -al` ``
Increment/decrement	Add or subtract 1	`$a++`
Logical	Boolean	`$a and $b`
String	Concatenation	`$a . $b`

Operators

Each operator takes a different number of operands:

- *Unary* operators, such as incrementing ($a++) or negation (!$a), take a single operand.

- *Binary* operators, which represent the bulk of PHP operators (including addition, subtraction, multiplication, and division), take two operands.

- The one *ternary* operator, which takes the form expr ? x : y, requires three operands. It's a terse, single-line if statement that returns x if expr is TRUE and y if expr is FALSE.

Associativity

Sometimes the expressions are processed from left to right, But some operators require processing from right to left, and this direction of processing is called the operator's *associativity*.

Operator	Description	Associativity
`< <= >= == != === !== <>`	Comparison	None
`!`	Logical `NOT`	Right
`~`	Bitwise `NOT`	Right
`++ --`	Increment and decrement	Right
`(int)`	Cast to an integer	Right
`(double) (float) (real)`	Cast to a floating-point number	Right

Associativity

Operator	Description	Associativity
`(string)`	Cast to a string	Right
`(array)`	Cast to an array	Right
`(object)`	Cast to an object	Right
`@`	Inhibit error reporting	Right
`= += -= *= /=`	Assignment	Right
`.= %= &= |= ^= <<= >>=`	Assignment	Right
`+`	Addition and unary plus	Left
`-`	Subtraction and negation	Left
`*`	Multiplication	Left

Associativity

Operator	Description	Associativity
/	Division	Left
%	Modulus	Left
.	String concatenation	Left
<< >> & ^ |	Bitwise	Left
?:	Ternary	Left
|| && and or xor	Logical	Left
,	Separator	Left

PHP's operators in order of precedence from high to low

Operator(s)	Type
()	Parentheses
++ --	Increment/decrement
!	Logical
* / %	Arithmetic
+ - .	Arithmetic and string
<< >>	Bitwise
< <= > >= <>	Comparison
== != === !==	Comparison
&	Bitwise (and references)

PHP's operators in order of precedence from high to low

Operator(s)	Type
^	Bitwise
|	Bitwise
&&	Logical
||	Logical
? :	Ternary
= += -= *= /= .= %= &= != ^= <<= >>=	Assignment
and	Logical
xor	Logical
or	Logical

Relational Operators

- Relational operators answer questions such as “Does this variable have a value of zero?” and “Which variable has a greater value?”
- These operators test two operands and return a Boolean result of either TRUE or FALSE.
- There are three types of relational operators:
  1. *equality*,
  2. *comparison*,
  3. and *logical*.

Equality == and inequality !=

- If the two operands of an equality expression are of different types, PHP will convert them to whatever type makes the best sense to it.

```php
<?php
    $a = "1000";
    $b = "+1000";
    $month = "March";
    if ($month == "March") echo "It's springtime";
    if ($a != $b) echo "2";
?>
```

Output:
It's springtime

Identity operator vs. equality

identity operator, which consists of three equals signs in a row, can be used to compare items without doing conversion.

```php
<?php
    $a = "1000";
    $b = "+1000";
    if ($a == $b) echo "1";
    if ($a === $b) echo "2";    // we have !== operator as well
?>
```

- The first if statement evaluated to TRUE. This is because both strings were first converted to numbers, and 1000 is the same numerical value as +1000.
- In contrast, the second if statement uses the identity operator, so it compares $a and $b as strings, sees that they are different, and thus doesn't output anything.

Identity operator vs. equality

```php
<?php
    $a = "1000";
    $b = "+1000";
    if ($a == $b) echo "1";
    if ($a === $b) echo "2";    // we have !== operator as well
?>
```

Output:
1

If we change the second if to **if** ($a !== $b) **echo** "2“ → The statement is asking whether $a and $b are *not* identical to each other in their actual string type, and the answer is TRUE; they are not the same.

Comparison operators

```php
<?php
    $a = 2; $b = 3;
    if ($a > $b) echo "$a is greater than $b<br>";
    if ($a < $b) echo "$a is less than $b<br>";
    if ($a >= $b) echo "$a is greater than or equal to $b<br>";
    if ($a <= $b) echo "$a is less than or equal to $b<br>";
?>
```

In this example, where $a is 2 and $b is 3, the following is output:

```
2 is less than 3
2 is less than or equal to 3
```

Logical operators

Logical operator	Description
AND	TRUE if both operands are TRUE
OR	TRUE if either operand is TRUE
XOR	TRUE if one of the two operands is TRUE
! (NOT)	TRUE if the operand is FALSE, or FALSE if the operand is TRUE

Possible variations of using the logical operators

Inputs		Operators and results		
a	b	AND	OR	XOR
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	TRUE
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE	FALSE

The OR operator can cause unintentional problems in if statements, because the second operand will not be evaluated if the first is evaluated as TRUE.

Logical operators

```php
<?php
    $a = 1; $b = 0;
    echo ($a AND $b) . "<br>";
    echo ($a or $b) . "<br>";
    echo ($a XOR $b) . "<br>";
    echo !$a . "<br>";
?>
```

Line by line, this example outputs:
nothing
1
1
nothing
meaning that only the second and third echo statements evaluate as TRUE. (Remember that NULL—or nothing—represents a value of FALSE.)

Null-safe Operator – php 8

- In the following example, the ?-> null-safe operator will short-circuit the remainder of the section if it encounters a null value and will return a null immediately without causing an error:

```
return $user->getAddress()?->getCountry()?->isoCode;
```

- Without this operator we would have needed to use several sequential calls to isset() for each section, testing them each in turn for having a null value. **match Expressions**

Conditionals

Conditionals alter program flow. They enable you to ask questions about certain things and respond to the answers you get in different ways.

1. if statement

```
<?php
    if ($bank_balance < 100){
        $money = 1000;
        $bank_balance += $money;
    }
?>
```

Conditionals

2. if… else

With an if...else statement, the first conditional statement is executed if the condition is TRUE. But if it’s FALSE, the second one is executed.

```
<?php
    if ($bank_balance < 100){
        $money = 1000;
        $bank_balance += $money;
    }
    else{
        $savings += 50;
        $bank_balance -= 50;
    }
?>
```

The elseif Statement

It is like an else statement, except that you place a further conditional expression prior to the conditional code.

It can be used when you want a number of different possibilities to occur, based upon a sequence of conditions.

```php
<?php
    if ($bank_balance < 100) {
        $money = 1000;
        $bank_balance += $money;
    }
    elseif ($bank_balance > 200){
        $savings += 100;
        $bank_balance -= 100;
    }
    else{
        $savings += 50;
        $bank_balance -= 50;
    }
?>
```

The elseif Statement

- An else statement closes either an if...else or an if...elseif...else statement. You can leave out a final else if it is not required, but you cannot have one before an elseif; you also cannot have an elseif before an if statement.

- You may have as many elseif statements as you like. But as the number of elseif statements increases, you would probably be better advised to consider a switch statement if it fits your needs.

The switch Statement

The switch statement is useful where one variable, or the result of an expression, can have multiple values, each of which should trigger a different activity.

```php
<?php
    if ($page == "Home") echo "You selected Home";
    elseif ($page == "About") echo "You selected About";
    elseif ($page == "News") echo "You selected News";
    elseif ($page == "Login") echo "You selected Login";
    elseif ($page == "Links") echo "You selected Links";
    else echo "Unrecognized selection";
?>
```

If we use a switch statement, the code in the previous example might look like

```php
<?php
switch ($page){
  case "Home":
     echo "You selected Home";
  break;
  case "About":
     echo "You selected About";
  break;
  case "News":
     echo "You selected News";
  break;
  case "Login":
     echo "You selected Login";
  break;
  case "Links":
     echo "You selected Links";
  break;
}
?>
```

Default action

A typical requirement in switch statements is to fall back on a default action if none of the case conditions are met.

For example, in the case of the previous example, you could add the following code immediately before the final curly brace.

```
default:
    echo "Unrecognized selection";
break;
```

match Expressions – PHP 8

A match expression is like a switch block, but it provides type-safe comparisons, supports a return value, does not require a break statement to break out, and also supports multiple matching values. So this rather cumbersome switch block.

match Expressions

```
switch ($country){
  case "UK":
  case "USA":
  case "Australia":
  default:
    $lang = "English";
    break;
  case "Spain":
    $lang = "Spanish";
    break;
  case "Germany":
  case "Austria":
    $lang = "German";
    break;
}
```

The switch can be replaced with the following much simpler match expression:

```
$lang = match($country)
{
"UK", "USA", "Australia", default => "English",
"Spain" => "Spanish",
"Germany", "Austria" => "German",
};
```

The ? (or ternary) Operator

- One way of avoiding the verbosity of if and else statements.
- It takes three operands rather than the typical two.

```php
<?php
    echo $fuel <= 1 ? "Fill tank now" : "There's enough fuel";
?>
```

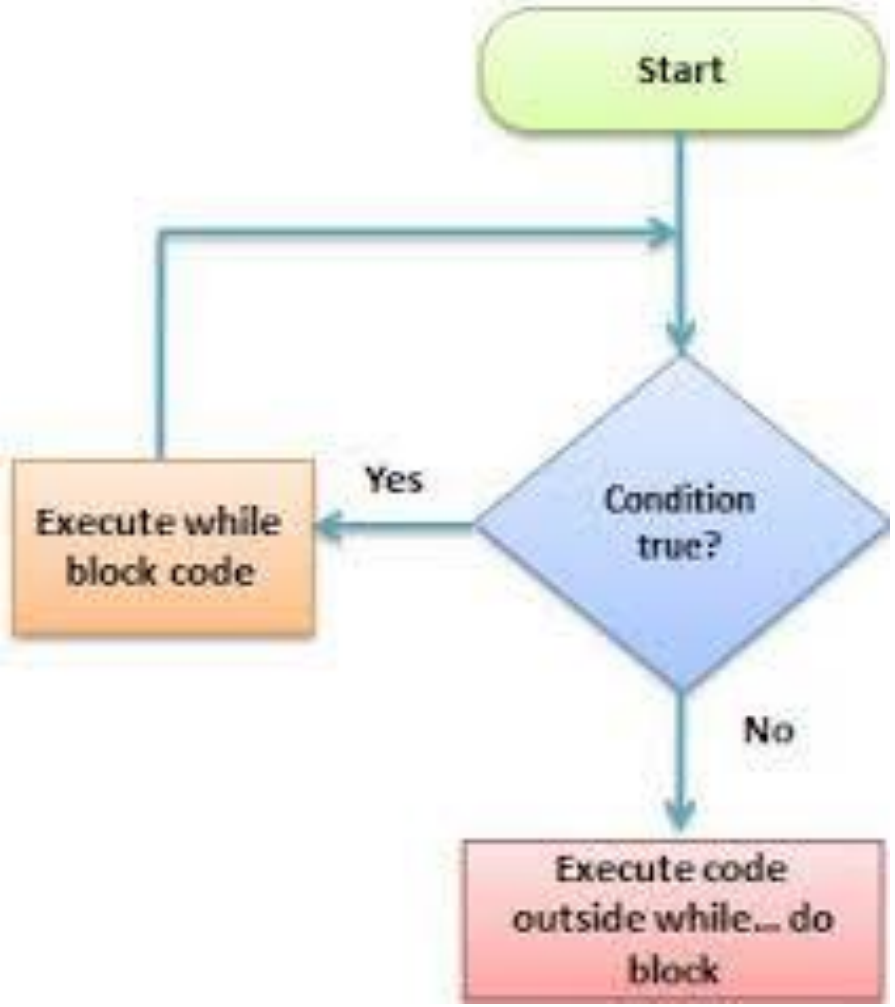
Looping

Loops are used when you want a program to repeat the same sequence of code again and again until something happens, such as a user inputting a value or reaching a natural end.

1. while
2. do..while
3. for

while loop

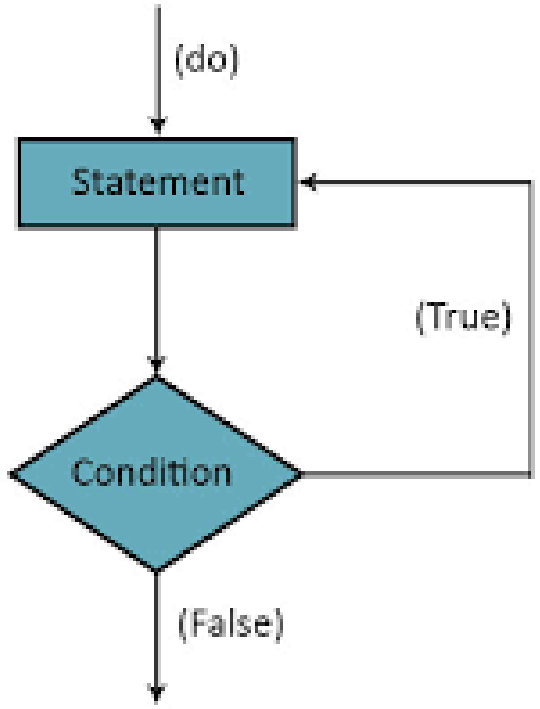
```php
<?php
  $count = 1;
  while ($count <= 12) {
    echo "$count times 12 is " . $count * 12 . "<br>";
    ++$count;
  }
?>
```



do...while Loops

It is used when you want a block of code to be executed at least once and made conditional only after that.

```php
<?php
    $count = 1;
    do {
        echo "$count times 12 is " . $count * 12;
        echo "<br>";
    } while (++$count <= 12);
?>
```



for Loops

```php
<?php
   for ($count = 1 ; $count <= 12 ; ++$count)
      echo "$count times 12 is " . $count * 12 . "<br>";
?>
```

Breaking Out of a Loop

- Just as you saw how to break out of a switch statement, you can also break out of a any loop using the same break command.
- This step can be necessary when, for example, one of your statements returns an error and the loop cannot continue executing safely.

The continue Statement

- The continue statement is a little like a break statement, except that it instructs PHP to stop processing the current iteration of the loop and move right to its next iteration.

- So, instead of breaking out of the whole loop, PHP exits only the current iteration.

```php
<?php
    $j = 11;
    while ($j > -10){
      $j--;
      if ($j == 0) continue;
      echo (10 / $j) . "<br>";
    }
?>
```

Output:

```
1
1.1111111111111
1.25
1.4285714285714
1.6666666666667
2
2.5
3.3333333333333
5
10
-10
-5
-3.3333333333333
-2.5
-2
-1.6666666666667
-1.4285714285714
-1.25
-1.1111111111111
-1
```

Implicit and Explicit Casting

- PHP is a loosely typed language that allows you to declare a variable and its type simply by using it. It also automatically converts values from one type to another whenever required. This is called *implicit casting*.
- However, at times PHP's implicit casting may not be what you want. Note that in the following example, the inputs to the division are integers. By default, PHP converts the output to floating point so it can give the most precise value—4.66 recurring.
- But what if we had wanted $c to be an integer instead?

```php
<?php
    $a = 56;
    $b = 12;
    $c = $a / $b;
    echo $c;
?>
```

Implicit and Explicit Casting

- In order to ensure that the value of $c in the previous example is cast to an integer, we place the expression within parentheses preceded by an int. This is called *explicit* casting.

```
$c = (int) ($a / $b);
```

- You can usually avoid having to use a cast by calling one of PHP's built-in functions. For example, to obtain an integer value, you could use the *intval* function.

References:

- Nixon, R. (2021) *Learning PHP, MySQL & JAVASCRIPT a step-by-step guide to creating dynamic websites*. S.l.: O'reilly Media. Chapter 4.