Cheatsheets / **Intermediate C#**

Learn C#: Loops

C# Arrays

In C#, an *array* is a structure representing a fixed length ordered collection of values or objects with the same type.
Arrays make it easier to organize and operate on large amounts of data. For example, rather than creating 100 integer variables, you can just create one array that stores all those integers!

```
// `numbers` array that stores integers
int[] numbers = { 3, 14, 59 };

// 'characters' array that stores strings
string[] characters = new string[] {
"Huey", "Dewey", "Louie" };
```

Declaring Arrays

A C# array variable is declared similarly to a non-array variable, with the addition of square brackets ([]) after the type specifier to denote it as an array.
The `new` keyword is needed when instantiating a new array to assign to the variable, as well as the array length in the square brackets. The array can also be instantiated with values using curly braces ({}). In this case the array length is not necessary.

```
// Declare an array of length 8 without
setting the values.
string[] stringArray = new string[8];

// Declare array and set its values to 3,
4, 5.
int[] intArray = new int[] { 3, 4, 5 };
```

Declare and Initialize array

In C#, one way an array can be declared and initialized at the same time is by assigning the newly declared array to a comma separated list of the values surrounded by curly braces ({}). Note how we can omit the type signature and `new` keyword on the right side of the assignment using this syntax. This is only possible during the array's declaration.

```
// `numbers` and `animals` are both
declared and initialized with values.
int[] numbers = { 1, 3, -10, 5, 8 };
string[] animals = { "shark", "bear",
"dog", "raccoon" };
```

codecademy

Array Element Access

In C#, the elements of an array are labeled incrementally, starting at 0 for the first element. For example, the 3rd element of an array would be indexed at 2, and the 6th element of an array would be indexed at 5.
A specific element can be accessed by using the square bracket operator, surrounding the index with square brackets. Once accessed, the element can be used in an expression, or modified like a regular variable.

```
// Initialize an array with 6 values.
int[] numbers = { 3, 14, 59, 26, 53, 0 };

// Assign the last element, the 6th
number in the array (currently 0), to 58.
numbers[5] = 58;

// Store the first element, 3, in the
variable `first`.
int first = numbers[0];
```

C# Array Length

The *Length* property of a C# array can be used to get the number of elements in a particular array.

```
int[] someArray = { 3, 4, 1, 6 };
Console.WriteLine(someArray.Length); //
Prints 4

string[] otherArray = { "foo", "bar",
"baz" };
Console.WriteLine(otherArray.Length); //
Prints 3
```

C# For Loops

A C# *for loop* executes a set of instructions for a specified number of times, based on three provided expressions. The three expressions are separated by semicolons, and in order they are:

- *Initializer*: This is run exactly once at the start of the loop, usually used to initialize the loop's iterator variable.
- *Conditional expression*: This boolean expression is checked before each iteration to see if it should run.
- *Iteration expression*: This is executed after each iteration of the loop and is usually used to update the iterator variable.

```
// This loop initializes i to 1, stops
looping once i is greater than 10, and
increases i by 1 after each iteration of
the loop
for (int i = 1; i <= 10; i++) {
  Console.WriteLine(i);
}

Console.WriteLine("Ready or not, here I
come!");
```

C# For Each Loop

A C# `foreach` loop runs a set of instructions once for each element in a given collection. For example, if an array has 200 elements, then the `foreach` loop's body will execute 200 times. At the start of each iteration, a variable is initialized to the current element being processed.
A *for each* loop is declared with the `foreach` keyword. Next, in parentheses, a *variable type* and *variable name* are followed by the `in` keyword and the collection to iterate over.

```
string[] states = { "Alabama", "Alaska", "Arizona", "Arkansas", "California", "Colorado" };

foreach (string state in states) {
  // The `state` variable takes on the value of an element in `states` and updates every iteration.
  Console.WriteLine(state);
}
// Will print each element of `states` in the order they appear in the array.
```

C# While Loop

In C#, a *while loop* executes a set of instructions continuously, as long as the given Boolean expression evaluates to `true`.
Note that the loop body might not run at all — since the Boolean condition is evaluated before every iteration of the *while loop*, including the very first, the loop will not execute if the conditional expression evaluates to `false` on the first pass.
The syntax to declare a while loop is simply the `while` keyword followed by a Boolean condition in parentheses.

```
string guess = "";
Console.WriteLine("What animal am I thinking of?");

// This loop will keep prompting the user, until they type in "dog".
while (guess != "dog") {
  Console.WriteLine("Make a guess: ");
  guess = Console.ReadLine().ToLower();
}

Console.WriteLine("That's right!");
```

C# Do While Loop

In C#, a *do...while* loop runs a set of instructions once and then continues running as long as the given boolean condition is `true`. Notice how this behavior is nearly identical to a *while* loop, with the distinction that a *do while* runs one or more times, whereas a *while* loop runs zero or more times.
The syntax to declare a *do while* is the `do` keyword, followed by the code block, then the `while` keyword with the Boolean condition in parentheses. Note that a semi-colon is required to end a *do...while* loop.

```
do
{
  Console.WriteLine("This will always run at least once, even if boolCondition is false to begin with.");
} while(boolCondition);

// whereas the do...while loop will always run at least once, a while loop may not run at all if its conditional expression is false to begin with
while(boolCondition)
{
  Console.WriteLine("If boolCondition is false to begin with, this code will not run at all.");
}
```

C# Infinite Loop

An *infinite loop* is a loop that never terminates because its conditional expression is always `true`. An *infinite loop* can be useful if a program consists of continuously executing one chunk of code. But, an unintentional *infinite loop* can cause a program to hang and become unresponsive due to being stuck in the loop.
A program running in a shell or terminal stuck in an infinite loop can be ended by terminating the process.

```
while (true) {
  // This will loop forever unless it contains some terminating statement such as `break`.
}
```

C# Jump Statements

Jump statements are tools used to give the programmer additional control over the program's control flow. They are very commonly used in the context of loops to exit from the loop early or skip parts of the loop's body.
Common control flow keywords include `break` and `continue`. The given code snippets provide examples of their usage.

```
while (true) {
  Console.WriteLine("This prints once.");
  // A `break` statement immediately terminates the loop that contains it.
  break;
}

for (int i = 1; i <= 10; i++) {
  // This prints every number from 1 through 10, except for 7.
  if (i == 7) {
    // A `continue` statement skips the rest of the loop and starts another iteration from the start.
    continue;
  }
  Console.WriteLine(i);
}
```

Print Share