Functions

C# Programming

In this chapter we will learn ...

- Why functions are useful
- How to define functions
- Passing data to functions as parameter arguments
- Returning data from a function
- Returning multiple items from a function using a tuple

Question

What are the benefits of functions?

Computer Science Principal - ALERT

DRY

- DO-NOT REPEAT YOURSELF

Put another way

- Don't duplicate code - It's more work to maintain it!
- You might forget all of the places you copied it to

So what is a function?

Functions ...

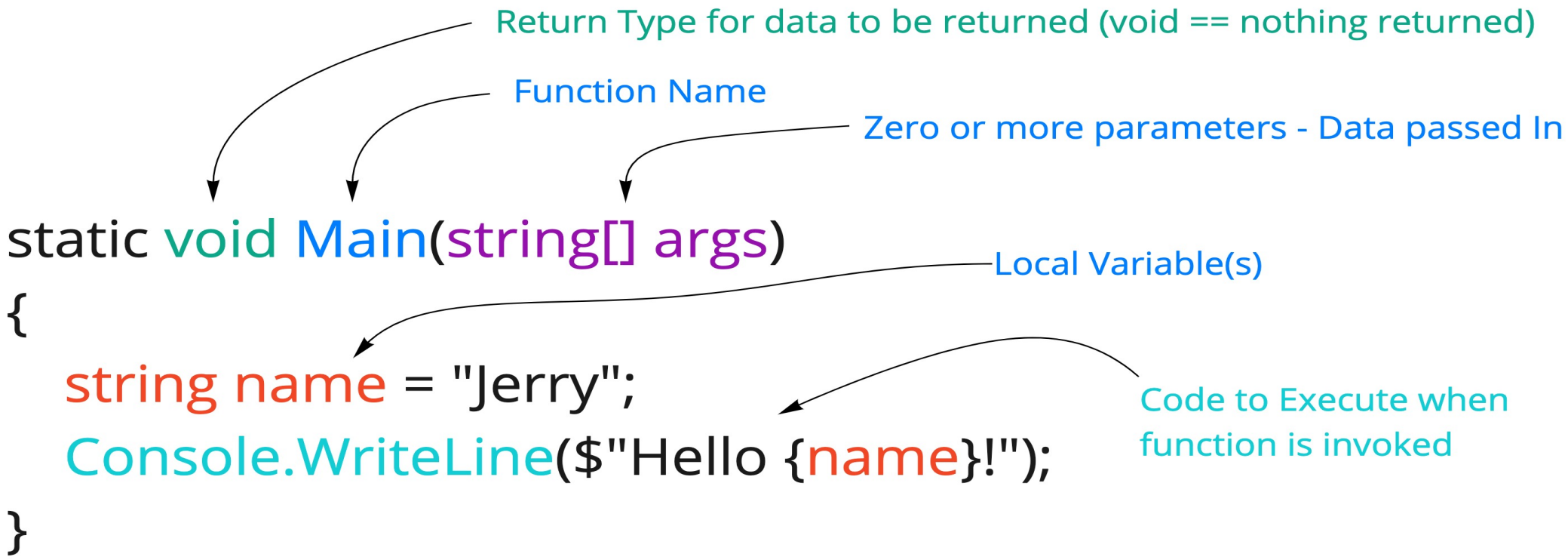
- Are re-usable, isolated, named/anonymous code blocks
- Can be called (invoked) - Makes the code within execute
- Can have data passed to them
- Can return data

C# Functions

Since C# 9.0 we have five types of functions

- Global Functions
  - Contained within one main module
- class/struct instance functions
  - Functions defined within a class or struct that require object instantiation
- static functions
  - Functions defined within a class or a struct that don't require object instantiation
- Local Functions
  - Functions defined within another function
- Lambda Functions
  - Concise expressions used as functional arguments to other functions

Structure of a C# Function



The "main" function

Is ... the starting point of a C# app is the Main(..) function

```
using System;

class TestClass {
    static void Main() {
        Console.WriteLine("Hello Jerry!");
    }
}
```

- Contained within a class
- Declared as **static**
- void - no value is returned from function

Top Level Module - C# 9.0

A single top level module acts as the Main function

- Only one top level module is allowed

```
using System;
Console.WriteLine("Hello World");
```

- Can contain local functions

Top Level Module with function(s)

```
using System;
string name = "Fred";
void SayHello() {
    Console.WriteLine($"Hello {name}");
}
name = "Tom";
SayHello();
```

- What is printed to the console (Fred or Tom)?
- What does "void" mean?
- How could you improve this function?

Dependent Functions

Making a function dependent on external/global variables

- Makes a function less independent
- External variables can change unexpectedly
- Impact of changing dependent variables is not clear to function caller
- Multithreaded applications would be more difficult to manage

Let's make functions independent

- Pass data to them
- Return data from them

Passing data to functions - Parameters

SayHello function has a parameter defined inside parentheses

- Function can have zero to many parameters

```
..
void SayHello(string name) {
    Console.WriteLine($"Hello {name}");
}
SayHello("Tom");
```

- When function is invoked an argument (value) is passed to the function's parameter
- Function executes with parameter's value set to argument

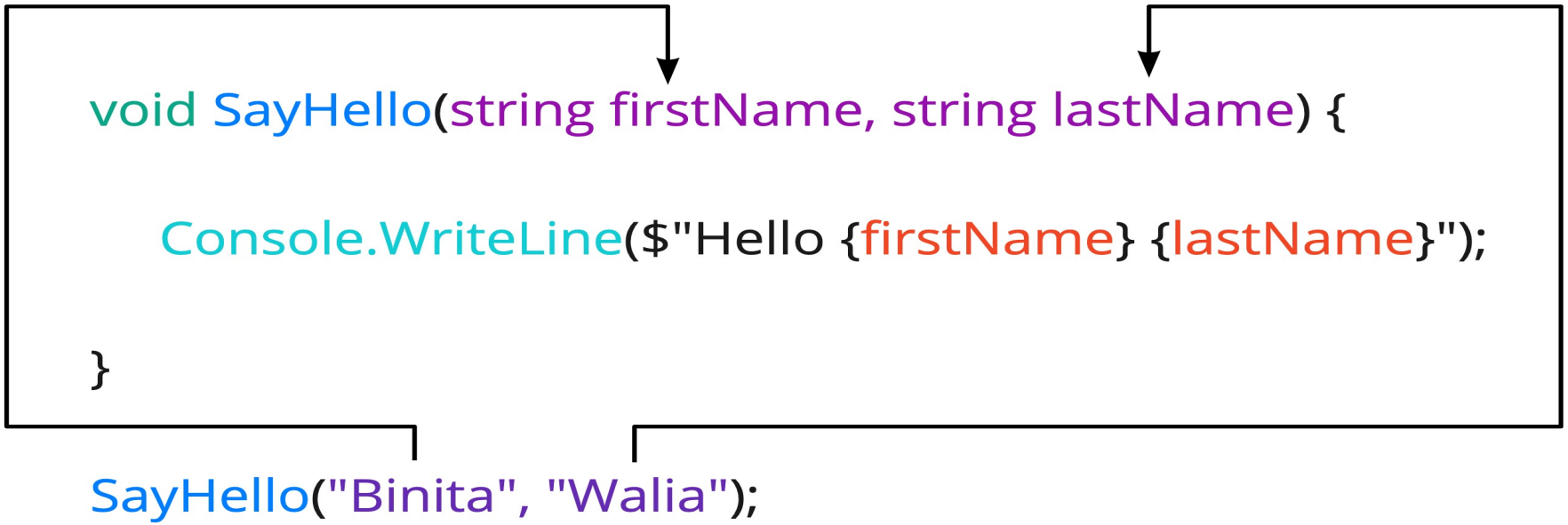
Passing arguments to function parameters

- Functions can have multiple parameters

Arguments can be passed to parameters

- By position - Arguments must be passed in the right order
- By name - Arguments can be passed in any order

Passing arguments by position



Named Function Arguments

Prefix argument with parameter name

- Can now pass arguments in any order

```
void SayHello(string firstName, string lastName) {

  Console.WriteLine($"Hello {firstName} {lastName}");

}

SayHello(lastName:"Walia", firstName:"Binita");
```

- Example shows lastName being passed in before firstName

Function Overloading

A function can have multiple implementations as long as

- Number of parameters or parameter types vary

Here we have two SayHello function implementations

- Correct function will execute base on passed parameters

```
void SayHello(string firstName, string lastName) {
  Console.WriteLine($"Hello {firstName} {lastName}");
}
void SayHello(string name) {
  Console.WriteLine($"Hi {name} ");
}
```

Optional parameters

Parameters can have a default value

- Makes passing an argument to parameter optional

```
void SayHello(string firstName, string lastName, string greeting = "Hello") {
  Console.WriteLine($"{greeting} {firstName} {lastName}");

}
SayHello("Fred", "Bloggs");
SayHello("Tasmanian", "Devil", "Hey!");
```

- greeting parameter has a default value of "Hello
    - If argument not supplied during function call default value is used

Variadic Parameters

- A single parameter can be passed multiple arguments
  - Must be rightmost parameter
  - Arguments do not need to be passed as an array

```
void SayHello(params string[] names) {
    foreach(string name in names) {
        Console.WriteLine($"Hi! {name}");
    }
}
SayHello("Fred", "Amy", "Bob"); // Correct
SayHello(new string[]{"Fred", "Amy", "Bob"}); // ERROR!
```

Read-Only Input Parameters

Prefixing a parameter declaration with in defines it as readonly

- readonly means it cannot be assigned to and changed

```
void SayHello(in string firstName, string lastName) {

    firstName = "Ooops"; // COMPILE ERROR! firstName is readonly variable
    lastName = "Double Ooops"; // All OK
    Console.WriteLine($"Hello {firstName} {lastName}");

}
```

Returning Data - return statement

- Functions can return data using **return** statement
  - Return any .NET type or custom type
  - Tuple

```
int Add(decimal lhs, decimal rhs) {
    int result = lhs + rhs;
    return result;
}
```

- Instead of void we declare the type of the return value ---> int

Why do we need to return data from functions?

1. Parameter values
    - are only accessible to executing code inside the function
2. Local variables (variables declared inside functions)
    - are only accessible to executing code inside the function

So to communicate the result of a function to the caller we need to

1. Return one or more values using **return**
2. Use an out parameter

Example

```
void Add(decimal lhs, decimal rhs) {
    int result = lhs + rhs;
}
Add(10, 5);
Console.WriteLine(result); // ERROR result does not exist in context!
```

Is fixed using a return statement and a compatible return type

```
int Add(decimal lhs, decimal rhs) {
    int result = lhs + rhs;
    return result;
}
int total = Add(10, 5);
Console.WriteLine(total); // OK! result is copied out to total variable
```

Defensive Coding Techniques

The following concepts promote more secure code:

- Immutability
- Failing Fast using contracts
- Validation

Immutability

An immutable object maintains it's state once initialized

- State cannot be changed

Benefit

- Using immutable objects establishes a consistent state
- Immutable objects can be shared across threads without locking
- Immutable objects are immune to race conditions
- An initialized state cannot be circumvented

Failing Fast using contracts

A key principal of security is defense in depth.

- If one security mechanism fails another level will succeed

Design by Contract

Requires the developer to define/uphold conditions for methods and function calls:

- Pre-Conditions - must be valid prior to execution. Caller must pass good data
- Post-Conditions - state guaranteed after execution or an exception will be thrown
- Class Invariants - assertions that establish properties and relationships are stable

A function caller must pass good data. The function must provide its service or throw an exception. If the data passed in is invalid then it will be rejected.

What to test

Methods (Functions that belong to class) and Constructors

- Test class invariants - test the state of instance variables are valid preconditions
- Test parameter pre-conditions
- Throw exceptions based on precondition failure

Global Functions

- Ideally should be atomic (don't rely on external scoped data)
- Test parameter pre-conditions
- Throw exceptions based on precondition failure

If the expected post-conditions of the function cannot be met after completion then it should throw an exception or expected return value

Design by Contract

Once the **caller** and the **function** understand the contract a more secure algorithm can be constructed:

```
void divide(int first, int second){
    // Acts as a guard condition
    if (second==0) throw new ArgumentException($"function divide->parameter->{nameof(second)} cannot be zero");

    return first / second;
}
```

- Can be implemented using assert statements (most languages allow asserts)
- System.Diagnostics.Contracts also provides Contract capabilities

Validation

Validation of data received from an insecure source is vital including

- User Input
- Untrusted Data Sources
  - Files
  - Databases
  - Data Streams

Validation

Steps to Validation

1. Validate the Data Source
    - Check the IP, checksum, signature
2. Normalize the data - (use String Normalize() function)
3. Check length/size/range of normalized data
4. Check syntax of data - Use pattern matching/ regular expressions if necessary
5. Check semantics - Does data make sense in context relative to other values
6. Check Domain - Does data exist as a valid member in the domain

Using Regular Expressions

Regular expressions allow you to define a pattern for what a string should look like

- A bit like when you use *.txt to search for all text files in a directory

Regex Tutorial

Using Regex in C#

Here we are validating a sort code \d is means any decimal number, {2} means two of them

```
using System.Text.RegularExpressions;
...
Regex r = new Regex(@"^\d{2}-\d{2}-\d{2}$");
string sortCode = "10-20-39";
bool isValid = r.IsMatch(sortCode);
```

In this chapter we learned ...

- Why functions are useful
- How to define functions
- Passing data to functions as parameter arguments
- Returning data from a function
- Returning multiple items from a function using a tuple