C# Basics

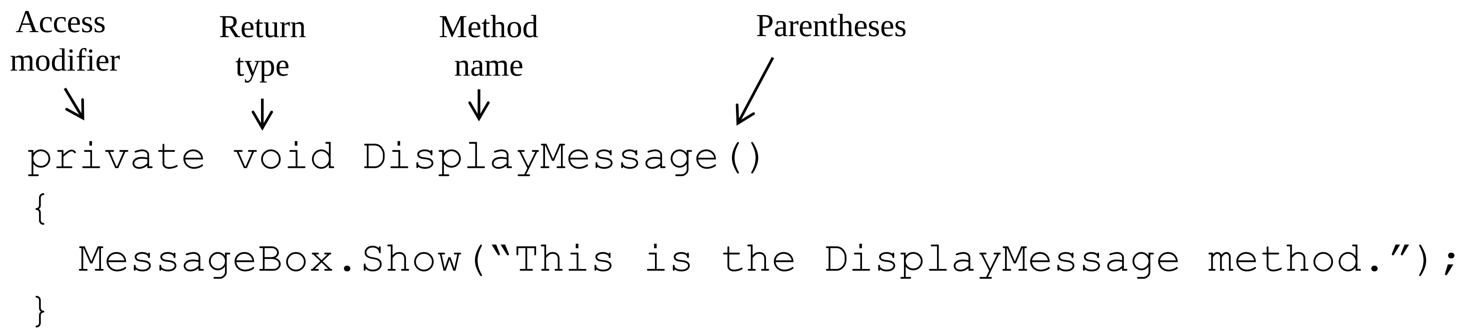
Methods

Introduction to Methods

- **Methods** can be used to break a complex program into small, manageable pieces
  - This approach is known as **divide and conquer**
  - In general terms, breaking down a program to smaller units of code, such as methods, is known as **modularization**
- Two types of methods are:
  - A **void** method simply executes a group of statements and then terminates
  - A **value-returning** method returns a value to the statement that called it

The Method Header

- A method header has the following parts :
  - **Access modifier**: keywords that defines the access control
    - private: a private method can be called only by code inside the same class as the method
    - public: a public method can be called by code that is outside the class.
  - **Return type**: specifies whether or not a method returns a value
  - **Method name**: the identifier of the method; must be unique in a given program. Use Pascal case
  - **Parentheses**: A method's name is always followed by a pair of parentheses



```
private void DisplayMessage()
{
  MessageBox.Show("This is the DisplayMessage method.");
}
```

Calling a Method

- A method executes when it is called
- Event handlers are called when specific events take place. Yet, methods are executed by **method call statements**.
- A method call statement is the name of the method followed by a pair of parentheses. For example,

- `DisplayMessage();`

```
private void goButton_Click(object sender, EventArgs e)
{
    MessageBox.Show("This is the goButton_Click method.");
    DisplayMessage();
}

private DisplayMessage()
{
  MessageBox.Show("This is the DisplayMessage method.");
}
```

Passing Arguments to Methods

- An **argument** is any piece of data that is passed into a method when the method is called
  - In the following, the statement calls the MessageBox.Show method and passes the string “Hello” as an argument:

```
MessageBox.Show(“Hello”);
```

- A **parameter** is a variable that receives an argument that is passed into a method
  - In the following, *value* is an int parameter:

```
private void DisplayValue(int value)
{
  MessageBox.Show(value.ToString());
}
```

- An argument’s data type must be assignment compatible with the receiving parameter’s data type

Methods Parameters

- Some functions have no parameters

```
public void CountUpdates() { … }
```

- Others can have several parameters
  - e.g., `float f0` and `float f1` below

```
public void PrintSum( float f0, float f1 ) {
   print( f0 + f1 );
}
```

- Parameters define the type and number of *arguments* that must be passed in when the function is called

```
PrintSum( 4f, 10.5f );   // Prints: "14.5"
```

The params Keyword

- `params` can be used to accept a variable number of similarly-typed parameters or an array of parameters

```
public float Sum( params float[] nums ) {
    float total = 0;
    foreach (float f in nums) {
        total += f;
    }
    return( total );
}

void Awake() {
    print( Sum( 1f ) );                              // Prints: "1f"
    print( Sum( 1f, 2f ) );                          // Prints: "3f"
    print( Sum( 1f, 2f, 3f ) );              // Prints: "6f"
    print( Sum( 1f, 2f, 3f, 4f ) );                  // Prints: "10f"
}
```

  – An array can also be passed into a params parameter

```
print( Sum( new float[] { 1f, 3.14f } ) );     // Prints: "4.14f"
```

Named Arguments

- C# allows you to specify which parameter an argument should be passed into. The syntax is:

  parameterName : *value*
- An argument that is written using this syntax is known as a **named argument**

```
private void showButton_Click(object sender, EventArgs e)
{
  showName(lastName : “Smith”, firstName : “Suzanne”);
}

private void ShowName(string firstName, string lastName)
{
  MessageBox.Show(firstName + “ “ + lastNmae);
}
```

- Notice that you get the same result if the call statement is:

```
showName(“Suzanne”, “Smith”);
```

Default Arguments

- C# allows you to provide a **default argument** for a method parameter

```
private void ShowTax(decimal price, decimal taxRate = 0.07m)
{
 decimal tax = price * taxRate;
}
```

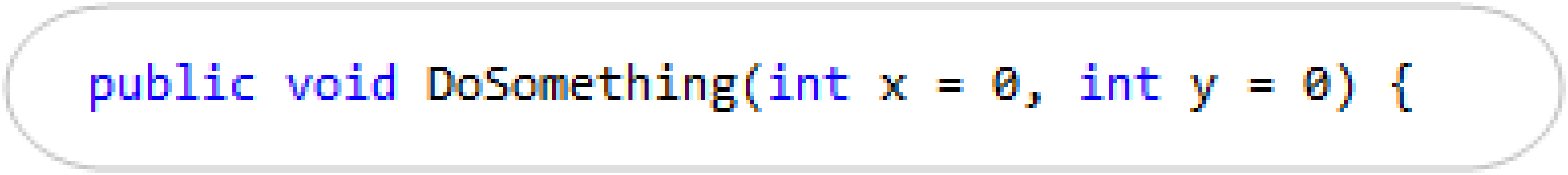
- The value of taxRate is defaulted to 0.07m. You can simply call the method by passing only the price. Parameters are optional when a default value is specified as part of a declaration.

```
showTax(100.0m);
```

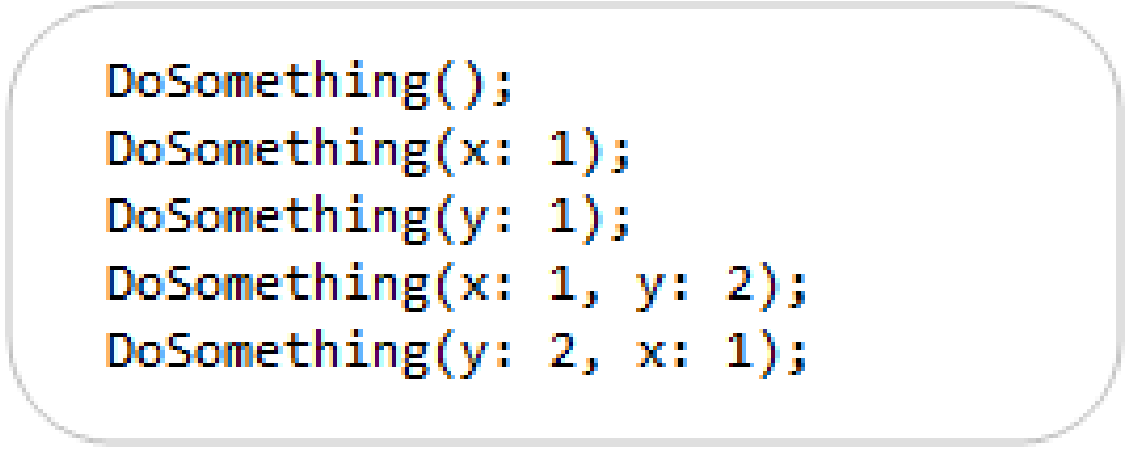
- You can also override the default argument

```
showTax(100.0m, 0.08m);
```

Named Arguments and Optional Parameters



```
public void DoSomething(int x = 0, int y = 0) {
```



```
DoSomething();
DoSomething(x: 1);
DoSomething(y: 1);
DoSomething(x: 1, y: 2);
DoSomething(y: 2, x: 1);
```

Passing Arguments by Reference

- A **reference parameter** is a special type of parameter that does not receive a copy of the argument's value
- It becomes a **reference** to the argument that was passed into it
- When an argument is passed by reference to a method, the method can change the value of the argument in the calling part of the program
- In C#, you declare a reference parameter by writing the **ref** keyword before the parameter variable's data type

```
private void SetToZero(ref int number)
{
  number =0;
}
```

- To call a method that has a reference parameter, you also use the keyword **ref** before the argument

```
int myVar = 99;
SetToZero(ref myVar);
```

The Return Statement

- There must be a **return** statement inside the method which is usually the last statement of the method. This return statement is used to return a value to the statement that called the method. For example,

```
private int sum(int num1, int num2)
{
  return num1 + num2;
}
```

- Notice that the returned value and the method's type must match
  - In the above example, the method is an int method, so it can only return int value

Returning Values

- Many methods return void

```
void Update() { … }
public void CountUpdates() { … }
```

- It's possible to return a single value from a method
  - The type of that value is the type of the method

```
public float Sum( float f0, float f1 ) {
    float f01 = f0 + f1;
    return( f01 );          // Returns the float f01
}

void Update() {
    float s = Sum( 3f, 0.14159f );
    print( s );             // Prints: "3.14159"
}
```

- A method can be declared with *any* return type!

```
public GameObject FindTheGameObject() { … }
```

Returning Values

- Sometimes, you want to use return even when the return type is void

```
public List<GameObject> reallyLongList;  // A List of many GObjs
public void MoveByName( string name, Vector3 loc ) {
    foreach (GameObject go in reallyLongList) {
        if (go.name == name) {
            go.transform.position = loc;
            return;   // Returns to avoid looping over the whole List
        }
    }
}
void Awake() {
    MoveByName( "Archon", Vector3.zero );
}
```

  – If "Phil" is the first GameObject in the List, returning could save lots of time!

Method Overloading

- The same function name can be defined several times with different parameters
- This is called *function overloading*

```
public float Sum( float f0, float f1 ) {
    return( f0 + f1 );
}

public Vector3 Sum( Vector3 v0, Vector3 v1 ) {
    return( v0 + v1 );
}

public Color Sum( Color c0, Color c1 ) {
    float r, g, b;
    r = Mathf.Min( c0.r + c1.r, 1f );  // Limits r to less than 1
    g = Mathf.Min( c0.g + c1.g, 1f );
    b = Mathf.Min( c0.b + c1.b, 1f );  // Because Color values
    a = Mathf.Min( c0.a + c1.a, 1f );  //   are between 0f and 1f
    return( new Color( r, g, b, a ) );
}
```

Methods Overloading

- Method signature: name and parameter types, number, and order (not the name, return type)

```
        int DoSomething (int a, int b) {}
```

The signature is DoSomthing(int, int)

- Overloading methods: have multiple methods with the same name, as long as the signatures are different

```
        void Foo (int x) {...}
        void Foo (double x) {...}
        void Foo (int x, float y) {...}
        void Foo (float x, int y) {...}

        float Foo (int x) {…}  // error
```

Naming Conventions

- Functions should always be named with CamelCaps
- Function names should always start with a capital letter

```
void Awake() { … }
void Start() { … }
public void PrintSum( float f0, float f1 ) { … }
public float Sum( float f0, float f1 ) { … }
public void MoveByName( string name, Vector3 loc ) { … }
```

- Declaring Method Inside a Class - Methods usually belong to a class