FUNCTIONS AND PARAMETERS

Topics

Topics

- Definition of a Function

Topics

- Definition of a Function
- Function Parameters and Arguments

Topics

- Definition of a Function
- Function Parameters and Arguments
- Returning Values

Topics

- Definition of a Function
- Function Parameters and Arguments
- Returning Values
  - Returning void

Topics

- Definition of a Function
- Function Parameters and Arguments
- Returning Values
  - Returning void
- Naming Conventions

Topics

- Definition of a Function
- Function Parameters and Arguments
- Returning Values
  - Returning void
- Naming Conventions
- Function Overloading

Topics

- Definition of a Function
- Function Parameters and Arguments
- Returning Values
  - Returning void
- Naming Conventions
- Function Overloading
- Optional Parameters

Topics

- Definition of a Function
- Function Parameters and Arguments
- Returning Values
  - Returning void
- Naming Conventions
- Function Overloading
- Optional Parameters
- The params Keyword

Topics

- Definition of a Function
- Function Parameters and Arguments
- Returning Values
  - Returning void
- Naming Conventions
- Function Overloading
- Optional Parameters
- The params Keyword
- Recursive Functions

Definition of a Function

Definition of a Function

- A function is a named set of actions

Definition of a Function

- A function is a named set of actions
- You've already used some functions

Definition of a Function

- A function is a named set of actions
- You've already used some functions

```
void Awake()  { … }      // Called when a GameObject begins
void Start()  { … }      // Called before the first Update()
void Update() { … }      // Called every frame
```

Definition of a Function

- A function is a named set of actions
- You've already used some functions

```
void Awake()  { … }      // Called when a GameObject begins
void Start()  { … }      // Called before the first Update()
void Update() { … }      // Called every frame
```

  – These were all built-in MonoBehaviour functions

Definition of a Function

- A function is a named set of actions
- You've already used some functions

```
void Awake()  { … }     // Called when a GameObject begins
void Start()  { … }     // Called before the first Update()
void Update() { … }     // Called every frame
```

  - These were all built-in MonoBehaviour functions
- Functions encapsulate action

Definition of a Function

- A function is a named set of actions
- You've already used some functions

```
void Awake()  { … }     // Called when a GameObject begins
void Start()  { … }     // Called before the first Update()
void Update() { … }     // Called every frame
```

  – These were all built-in MonoBehaviour functions
- Functions encapsulate action
- Functions have their own *scope*

Definition of a Function

- A function is a named set of actions
- You've already used some functions

```
void Awake()  { … }     // Called when a GameObject begins
void Start()  { … }     // Called before the first Update()
void Update() { … }     // Called every frame
```

  – These were all built-in MonoBehaviour functions

- Functions encapsulate action
- Functions have their own *scope*
  – Variables declared within a function are scoped to that function and cease to exist when it completes

Definition of a Function

- A function is a named set of actions
- You've already used some functions

```
void Awake()  { … }     // Called when a GameObject begins
void Start()  { … }     // Called before the first Update()
void Update() { … }     // Called every frame
```

  – These were all built-in MonoBehaviour functions

- Functions encapsulate action
- Functions have their own *scope*
  – Variables declared within a function are scoped to that function and cease to exist when it completes
- Now, you will write functions of your own, and you will choose when to call them

THE PUBLIC KEYWORD

THE PUBLIC KEYWORD

- Each C# script contains at least one class

THE PUBLIC KEYWORD

- Each C# script contains at least one class
  - A class is a collection of both variables and functions

THE PUBLIC KEYWORD

- Each C# script contains at least one class
  - A class is a collection of both variables and functions
- In these slides, you will see the `public` keyword used before variable and function names

THE PUBLIC KEYWORD

- Each C# script contains at least one class
  - A class is a collection of both variables and functions
- In these slides, you will see the `public` keyword used before variable and function names

```
public int counter = 0;
public void CountUpdates() { … }
```

THE PUBLIC KEYWORD

- Each C# script contains at least one class
  - A class is a collection of both variables and functions
- In these slides, you will see the `public` keyword used before variable and function names

```
public int counter = 0;
public void CountUpdates() { … }
```

- Public elements are scoped to the entire class

THE PUBLIC KEYWORD

- Each C# script contains at least one class
  - A class is a collection of both variables and functions
- In these slides, you will see the `public` keyword used before variable and function names

```
public int counter = 0;
public void CountUpdates() { … }
```

- Public elements are scoped to the entire class
  - Every function in the class can see them

THE PUBLIC KEYWORD

- Each C# script contains at least one class
  - A class is a collection of both variables and functions
- In these slides, you will see the `public` keyword used before variable and function names

```
    public int counter = 0;
    public void CountUpdates() { … }
```

- Public elements are scoped to the entire class
  - Every function in the class can see them
- Public variables and functions are also visible to other classes that encounter this class

THE PUBLIC KEYWORD

- Each C# script contains at least one class
  - A class is a collection of both variables and functions
- In these slides, you will see the `public` keyword used before variable and function names

```
public int counter = 0;
public void CountUpdates() { … }
```

- Public elements are scoped to the entire class
  - Every function in the class can see them
- Public variables and functions are also visible to other classes that encounter this class
- Unity functions (like `Update()`) are automatically public (though they lack the `public` keyword)

Definition of a Function

Definition of a Function

- A simple function example

Definition of a Function

- A simple function example

```
public class FunctionExample : MonoBehaviour {
    public int counter = 0;                                     // 1
```

Definition of a Function

- A simple function example

```
public class FunctionExample : MonoBehaviour {
    public int counter = 0;                                 // 1

    void Update()  {
        counter++;                                          // 2
        CountUpdates();                                     // 3
    }
```

Definition of a Function

- A simple function example

```
public class FunctionExample : MonoBehaviour {
    public int counter = 0;                                 // 1

    void Update()  {
        counter++;                                          // 2
        CountUpdates();                                     // 3
    }

    public void CountUpdates() {                            // 4
        string str = "Updates: "+counter;                   // 5
        print( str );                                       // 6
    }
}
```

Definition of a Function

- A simple function example

```
public class FunctionExample : MonoBehaviour {
    public int counter = 0;                               // 1

    void Update()  {
        counter++;                                        // 2
        CountUpdates();                                   // 3
    }

    public void CountUpdates() {                          // 4
        string str = "Updates: "+counter;                 // 5
        print( str );                                     // 6
    }
}
```

  – CountUpdates() is *called* by Update() every frame

Definition of a Function

- A simple function example

```
public class FunctionExample : MonoBehaviour {
    public int counter = 0;                                  // 1

    void Update()  {
        counter++;                                           // 2
        CountUpdates();                                      // 3
    }

    public void CountUpdates() {                             // 4
        string str = "Updates: "+counter;                    // 5
        print( str );                                        // 6
    }
}
```

  - CountUpdates() is *called* by Update() every frame
  - What will this function do at each numbered point (`// #`)?

Definition of a Function

- A simple function example

```
public class FunctionExample : MonoBehaviour {
    public int counter = 0;                                // 1

    void Update()  {
        counter++;                                         // 2
        CountUpdates();                                    // 3
    }

    public void CountUpdates() {                           // 4
        string str = "Updates: "+counter;                  // 5
        print( str );                                      // 6
    }
}
```

  - CountUpdates() is *called* by Update() every frame
  - What will this function do at each numbered point (`// #`)?
  - What is the scope of `counter`?

Definition of a Function

- A simple function example

```
public class FunctionExample : MonoBehaviour {
    public int counter = 0;                                    // 1

    void Update()  {
        counter++;                                             // 2
        CountUpdates();                                        // 3
    }

    public void CountUpdates() {                               // 4
        string str = "Updates: "+counter;                      // 5
        print( str );                                          // 6
    }
}
```

  - CountUpdates() is *called* by Update() every frame
  - What will this function do at each numbered point (`// #`)?
  - What is the scope of `counter`?
  - What is the scope of `str`?

Function Parameters and Arguments

Function Parameters and Arguments

- Some functions have no parameters

Function Parameters and Arguments

- Some functions have no parameters

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

Function Parameters and Arguments

- Some functions have no parameters

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

- Others can have several parameters

Function Parameters and Arguments

- Some functions have no parameters

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

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

Function Parameters and Arguments

- 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 );
}
```

Function Parameters and Arguments

- 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

Function Parameters and Arguments

- 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"
```

Returning Values

Returning Values

- Most functions we've seen return void

Returning Values

- Most functions we've seen return void

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

Returning Values

- Most functions we've seen return void

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

- It's possible to return a single value from a function

Returning Values

- Most functions we've seen return void

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

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

Returning Values

- Most functions we've seen return void

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

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

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

PEARSON

Returning Values

- Most functions we've seen return void

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

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

```
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"
}
```

Returning Values

- Most functions we've seen return void

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

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

```
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 function can be declared with *any* return type!

Returning Values

- Most functions we've seen return void

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

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

```
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 function can be declared with *any* return type!

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

Returning Values

Returning Values

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

Returning Values

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

```
public List<GameObject> reallyLongList; // A List of many GObjs
```

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
        }
    }
}
```

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 );
}
```

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!

Naming Conventions

Naming Conventions

- Functions should always be named with CamelCaps

Naming Conventions

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

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 ) { … }
```

Function Overloading

Function Overloading

- **The same function name can be defined several times with different parameters**

Function Overloading

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

Function 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 );
}
```

Function 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 );
}
```

Function 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 ) );
}
```

Optional Parameters

Optional Parameters

- Some parameters can have default values

Optional Parameters

- Some parameters can have default values
  - Optional parameters must come *after* required parameters

Optional Parameters

- Some parameters can have default values
  - Optional parameters must come *after* required parameters

```
public void SetX( GameObject go, float x = 0f ) {
    Vector3 tempPos = go.transform.position;
    tempPos.x = x;
    go.transform.position = tempPos;
}
```

Optional Parameters

- Some parameters can have default values
  - Optional parameters must come *after* required parameters

```
public void SetX( GameObject go, float x = 0f ) {
    Vector3 tempPos = go.transform.position;
    tempPos.x = x;
    go.transform.position = tempPos;
}

void Awake() {
    SetX( this.gameObject, 25f ); // Moves gameObject to x=25f
```

Optional Parameters

- Some parameters can have default values
  - Optional parameters must come *after* required parameters

```
public void SetX( GameObject go, float x = 0f ) {
    Vector3 tempPos = go.transform.position;
    tempPos.x = x;
    go.transform.position = tempPos;
}

void Awake() {
    SetX( this.gameObject, 25f ); // Moves gameObject to x=25f
    SetX( this.gameObject );      // Moves gameObject to x=0f
}
```

The `params` Keyword

The `params` Keyword

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

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 );
}
```

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"
}
```

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

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"
```

Recursive Functions

Recursive Functions

- Some functions are designed to call themselves repeatedly

Recursive Functions

- Some functions are designed to call themselves repeatedly

```
public int Factorial( int n ) {
     if (n < 0)   return( 0 );    // if statements can be just 1 line
     if (n == 0) return( 1 );     // This is the terminal case
```

Recursive Functions

- Some functions are designed to call themselves repeatedly

```
public int Factorial( int n ) {
    if (n < 0)  return( 0 );    // if statements can be just 1 line
    if (n == 0) return( 1 );    // This is the terminal case

    int result = n * Fac(n-1);  // This is the recurring case
```

Recursive Functions

- Some functions are designed to call themselves repeatedly

```
public int Factorial( int n ) {
     if (n < 0)  return( 0 );     // if statements can be just 1 line
     if (n == 0) return( 1 );     // This is the terminal case

     int result = n * Fac(n-1);   // This is the recurring case

     return( result );            // Return the final result
}
```

Recursive Functions

- Some functions are designed to call themselves repeatedly

```
public int Factorial( int n ) {
    if (n < 0)  return( 0 );     // if statements can be just 1 line
    if (n == 0) return( 1 );     // This is the terminal case

    int result = n * Fac(n-1);   // This is the recurring case

    return( result );            // Return the final result
}

void Awake() {
    print( Fac( -1 ) );                  // Prints: "0"
    print( Fac(  0 ) );                  // Prints: "1"
    print( Fac(  5 ) );                  // Prints: "120"
}
```

Recursive Functions

- Some functions are designed to call themselves repeatedly

```
public int Factorial( int n ) {
    if (n < 0)   return( 0 );     // if statements can be just 1 line
    if (n == 0) return( 1 );      // This is the terminal case

    int result = n * Fac(n-1);    // This is the recurring case

    return( result );             // Return the final result
}

void Awake() {
    print( Fac( -1 ) );                      // Prints: "0"
    print( Fac(  0 ) );                      // Prints: "1"
    print( Fac(  5 ) );                      // Prints: "120"
}
```

  – Fac(5) will call itself recursively until it gets to the terminal case of Fac(0) and then start returning values

Recursive Functions

PEARSON

Recursive Functions

- `Fac(5)` will call itself recursively until it reaches the terminal case `Fac(0)` and then start returning values

Recursive Functions

- `Fac(5)` will call itself recursively until it reaches the terminal case `Fac(0)` and then start returning values
  - The chain of recursion would look something like this

Recursive Functions

- `Fac(5)` will call itself recursively until it reaches the terminal case `Fac(0)` and then start returning values
  - The chain of recursion would look something like this

```
Fac(5)
5 * Fac(4)
5 * 4 * Fac(3)
5 * 4 * 3 * Fac(2)
5 * 4 * 3 * 2 * Fac(1)
5 * 4 * 3 * 2 * 1 * Fac(0)
5 * 4 * 3 * 2 * 1 * 1
5 * 4 * 3 * 2 * 1
5 * 4 * 3 * 2
5 * 4 * 6
5 * 24
120
```

Chapter 23 – Summary

Chapter 23 – Summary

- Functions are named collections of actions

Chapter 23 – Summary

- Functions are named collections of actions
- Functions can define *parameters* that must be passed in as *arguments* & can return a single value

Chapter 23 – Summary

- Functions are named collections of actions
- Functions can define *parameters* that must be passed in as *arguments* & can return a single value
- Functions are named with uppercase CamelCaps

Chapter 23 – Summary

- Functions are named collections of actions
- Functions can define *parameters* that must be passed in as *arguments* & can return a single value
- Functions are named with uppercase CamelCaps
- Functions can be overloaded to act differently based on different input argument types and numbers

Chapter 23 – Summary

- Functions are named collections of actions
- Functions can define *parameters* that must be passed in as *arguments* & can return a single value
- Functions are named with uppercase CamelCaps
- Functions can be overloaded to act differently based on different input argument types and numbers
- Some parameters can be optional

Chapter 23 – Summary

- Functions are named collections of actions
- Functions can define *parameters* that must be passed in as *arguments* & can return a single value
- Functions are named with uppercase CamelCaps
- Functions can be overloaded to act differently based on different input argument types and numbers
- Some parameters can be optional
- The params keyword allows variable numbers of arguments

Chapter 23 – Summary

- Functions are named collections of actions
- Functions can define *parameters* that must be passed in as *arguments* & can return a single value
- Functions are named with uppercase CamelCaps
- Functions can be overloaded to act differently based on different input argument types and numbers
- Some parameters can be optional
- The params keyword allows variable numbers of arguments
- Recursive functions are designed to call themselves

Chapter 23 – Summary

- Functions are named collections of actions
- Functions can define *parameters* that must be passed in as *arguments* & can return a single value
- Functions are named with uppercase CamelCaps
- Functions can be overloaded to act differently based on different input argument types and numbers
- Some parameters can be optional
- The params keyword allows variable numbers of arguments
- Recursive functions are designed to call themselves
- Next Chapter: Learn about debugging!