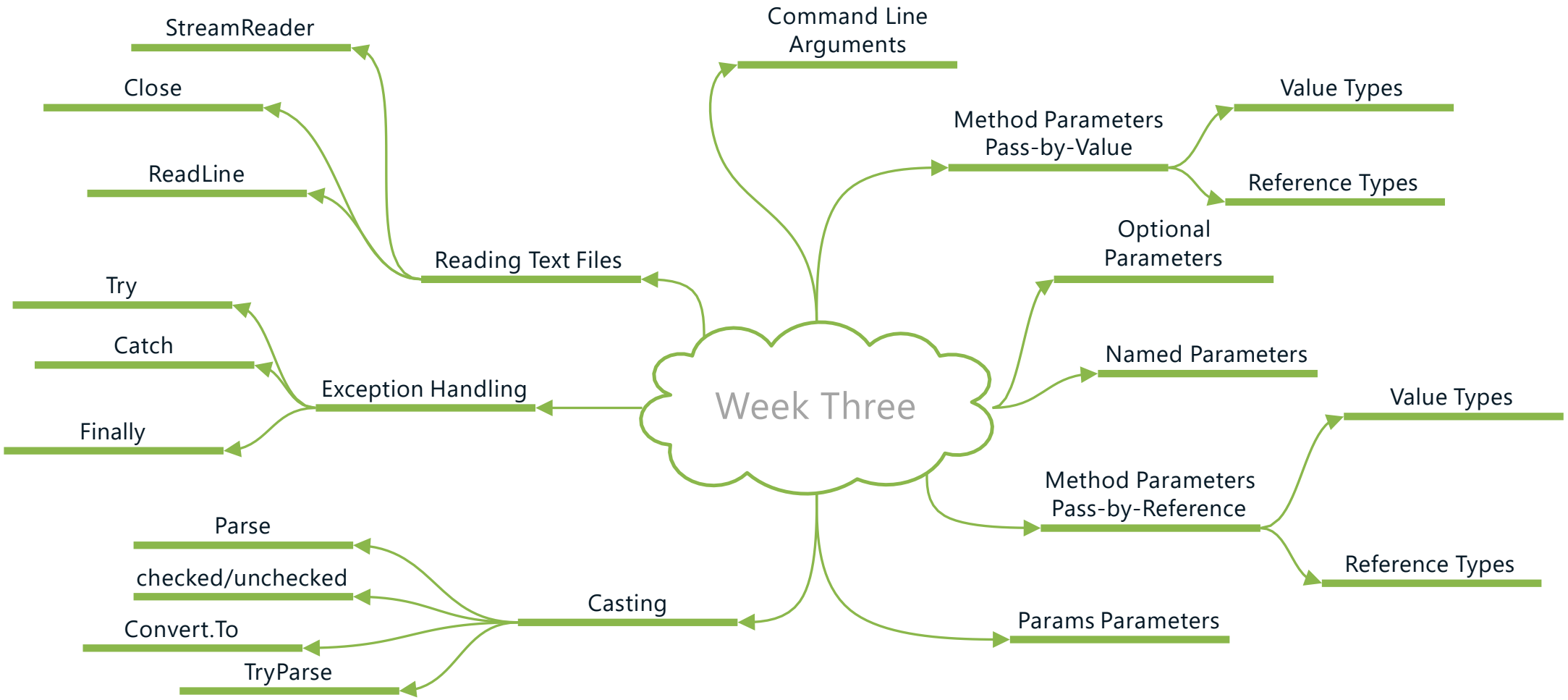
COMP 3602

C# Application Development

Week Three

Lake

Tonight's Learning Outcomes



Value Type – Pass by Value

```
0 references
15  static void Main(string[] args)
16  {
17      Console.Title = "Value Type Pass By Value Example";
18
19      int testValue = 1750;
20
21      Console.WriteLine($"{labelOriginal, -36} {testValue, 6:N0}");
22
23      add50PassByValue(testValue);
24
25      Console.WriteLine($"{labelAfter, -36} {testValue, 6:N0}\n\n");
26  }
27
    1 reference
28  private static void add50PassByValue(int input)
29  {
30      input += 50;
31      Console.WriteLine($"{labelInside, -36} {input, 6:N0}");
32  }
33
```

```
Value Type Pass By Value Example
Original value:                          1,750
Value inside PassByValue method:         1,800
Value after PassByValueCall:             1,750
```

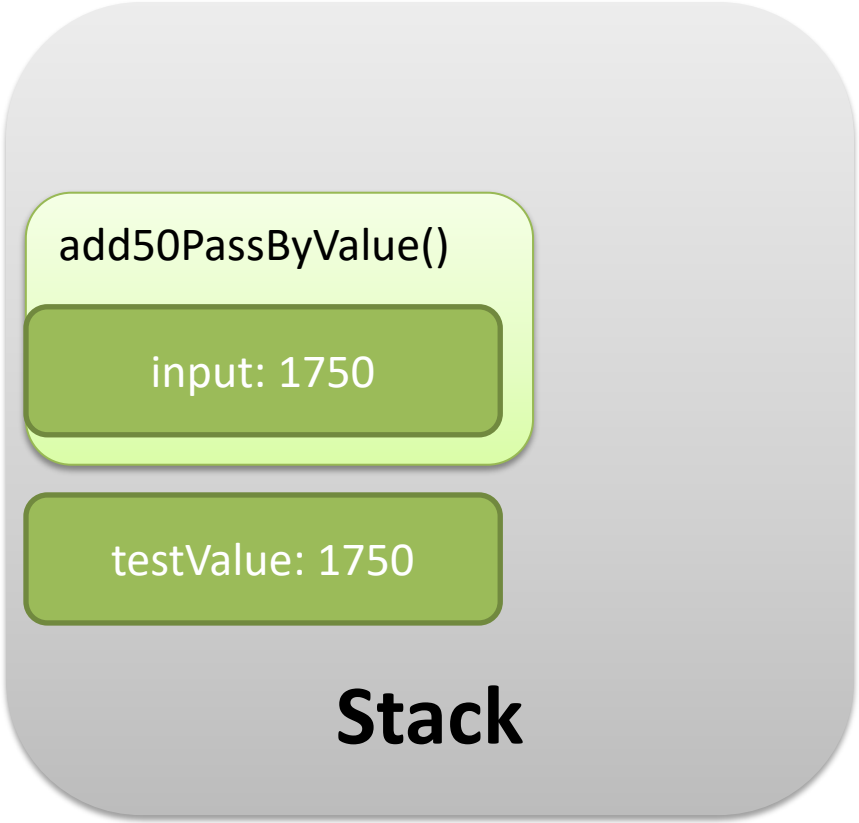
- All method parameters are passed by-value by default.
- **A *copy* of the parameter is made and passed to the method.**
- With value types this means the actual value is copied so the original value is unaffected by actions taken in the method.

Value Type – Pass by Value

```
Console.Title = "Value Type Pass By Value Example";

int testValue = 1750;
```

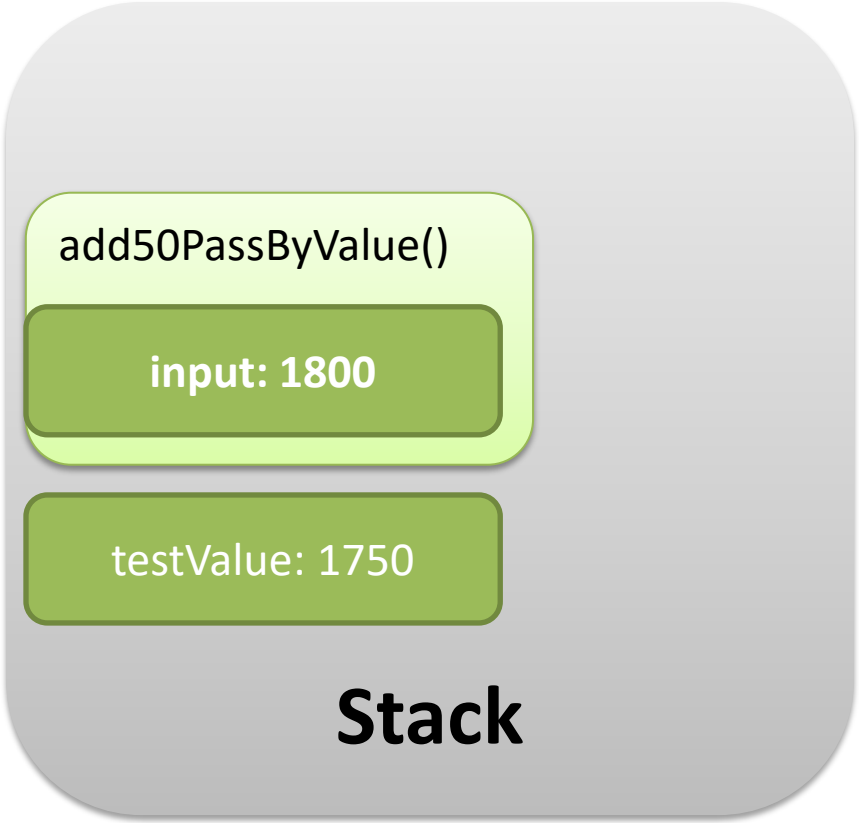
Value Type – Pass by Value

```
1 reference
private static void add50PassByValue(int input)
{
    input += 50;
    Console.WriteLine($"{labelInside,-36} {input,6:N0}");
}
```



Value Type – Pass by Value

```
1 reference
private static void add50PassByValue(int input)
{
    input += 50;
    Console.WriteLine($"{labelInside,-36} {input,6:N0}");
}
```



Value Type – Pass by Reference

```
0 references
static void Main(string[] args)
{
    Console.Title = "Value Type Pass By Reference Example";

    int testValue = 1750;

    Console.WriteLine($"{labelOriginal, -36} {testValue, 6:N0}");

    add50PassByReference(ref testValue);

    Console.WriteLine($"{labelAfter, -36} {testValue, 6:N0}\n\n");
}

1 reference
private static void add50PassByReference(ref int input)
{
    input += 50;
    Console.WriteLine($"{labelInside, -36} {input, 6:N0}");
}
```

```
Value Type Pass By Reference Example
Original value:                          1,750
Value inside PassByReference method:     1,800
Value after PassByReferenceCall:         1,800
```

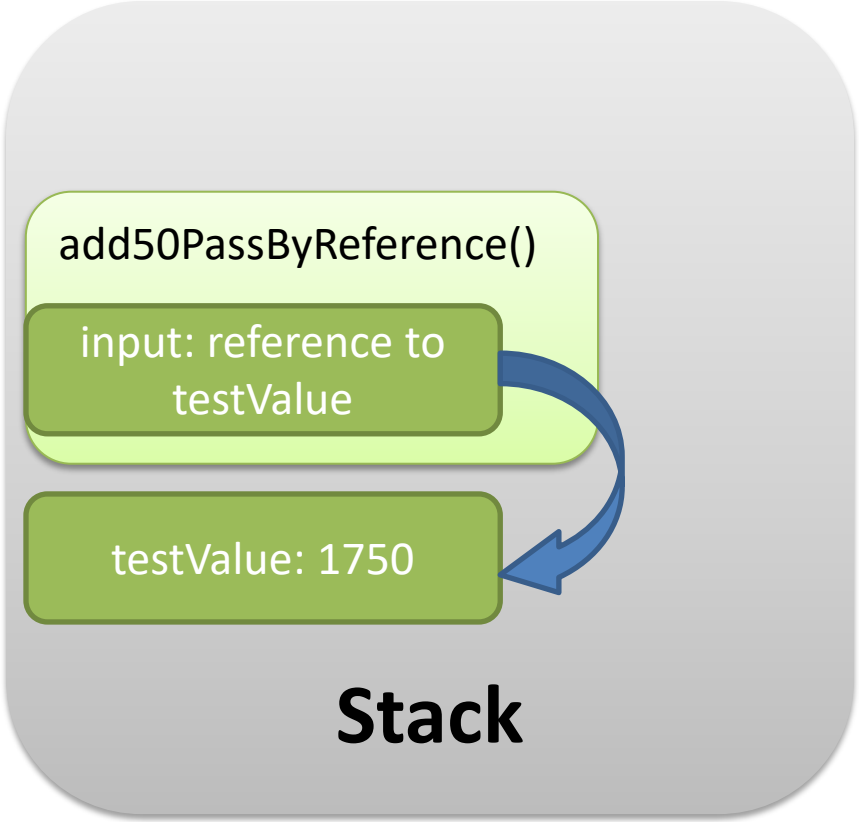
- Using the ref modifier passes a parameter by-reference
- A reference to the original value is passed so any actions taken in the method are permanent.
- The ref modifier is used both in the method definition and the call site.

Value Type – Pass by Reference

```
static void Main(string[] args)
{
    Console.Title = "Value Type Pass By Reference Example";

    int testValue = 1750;
```

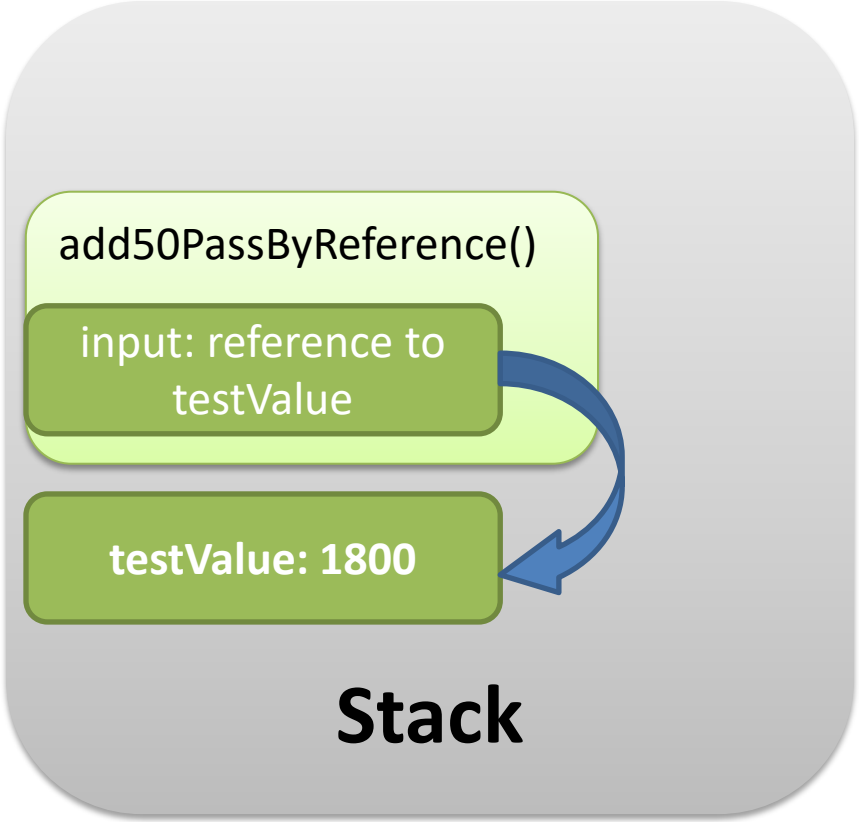
Value Type – Pass by Reference

```
private static void add50PassByReference(ref int input)
{
    input += 50;
    Console.WriteLine($"{labelInside,-36} {input,6:N0}");
}
```



Value Type – Pass by Reference

```
private static void add50PassByReference(ref int input)
{
    input += 50;
    Console.WriteLine($"{labelInside,-36} {input,6:N0}");
}
```



Reference Type – Pass by Reference

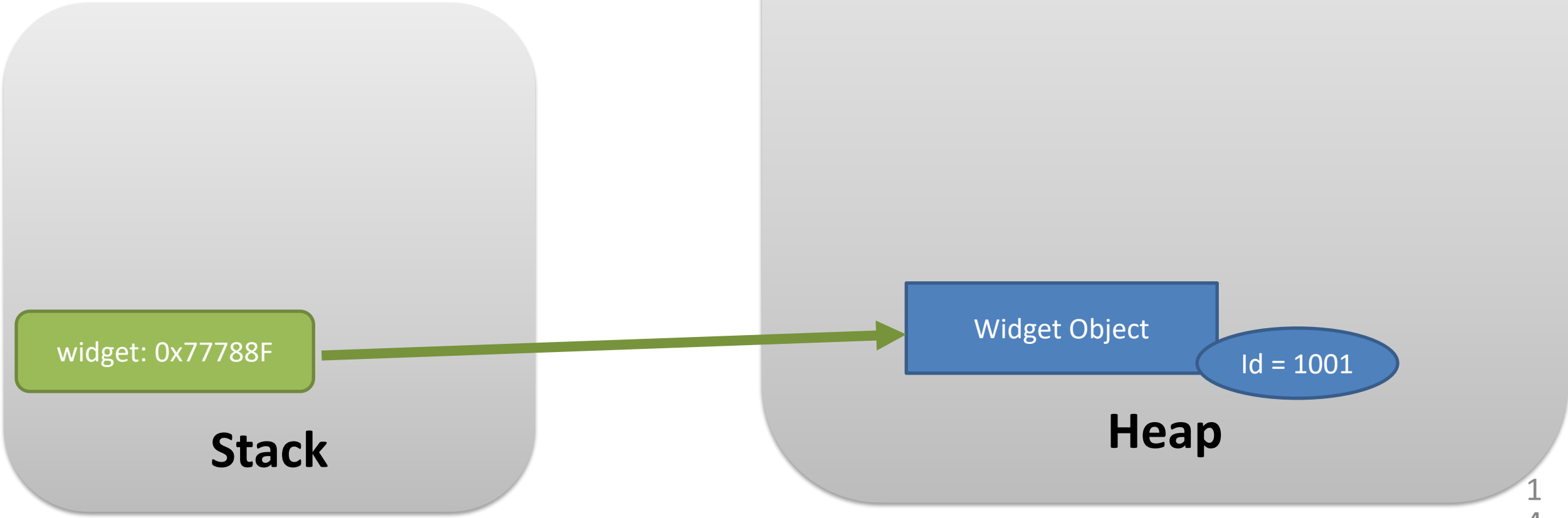
```
0 references
static void Main(string[] args)
{
    Console.Title = "Reference Type Pass By Reference Example";

    Widget widget = new Widget { Id = 1001 };

    Console.WriteLine($"{labelOriginal, -36} {widget.Id, 6:N0}");

    add50PassByReference(ref widget);

    Console.WriteLine($"{labelAfter, -36} {widget.Id, 6:N0}\n\n");
}

1 reference
private static void add50PassByReference(ref Widget input)
{
    input.Id += 50;
    Console.WriteLine($"{labelInside, -36} {input.Id, 6:N0}");
}
```

```
Reference Type Pass By Reference Example
Original value:                          1,001
Value inside PassByReference method:     1,051
Value after PassByReferenceCall:         1,051
```

- A reference to the reference is passed to the method.
- Passed reference to reference still refers to the original value.
- Any actions taken in the method are permanent.

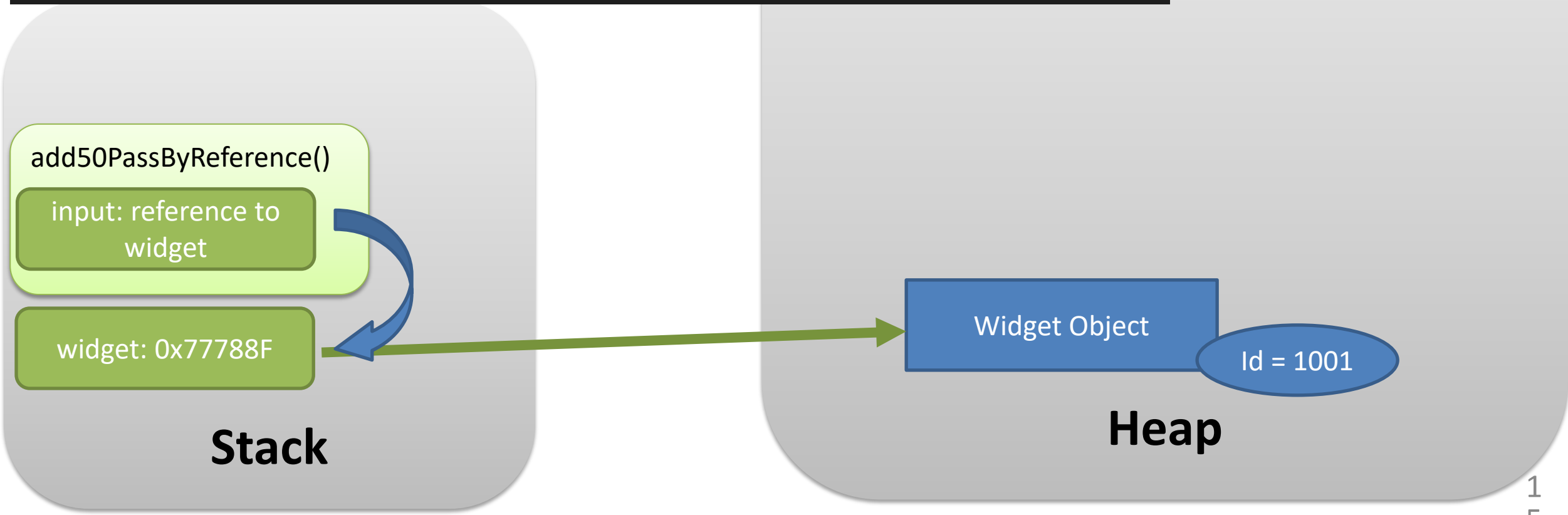
Reference Type – Pass by Reference

```
static void Main(string[] args)
{
    Console.Title = "Reference Type Pass By Reference Example";

    Widget widget = new Widget { Id = 1001 };
```



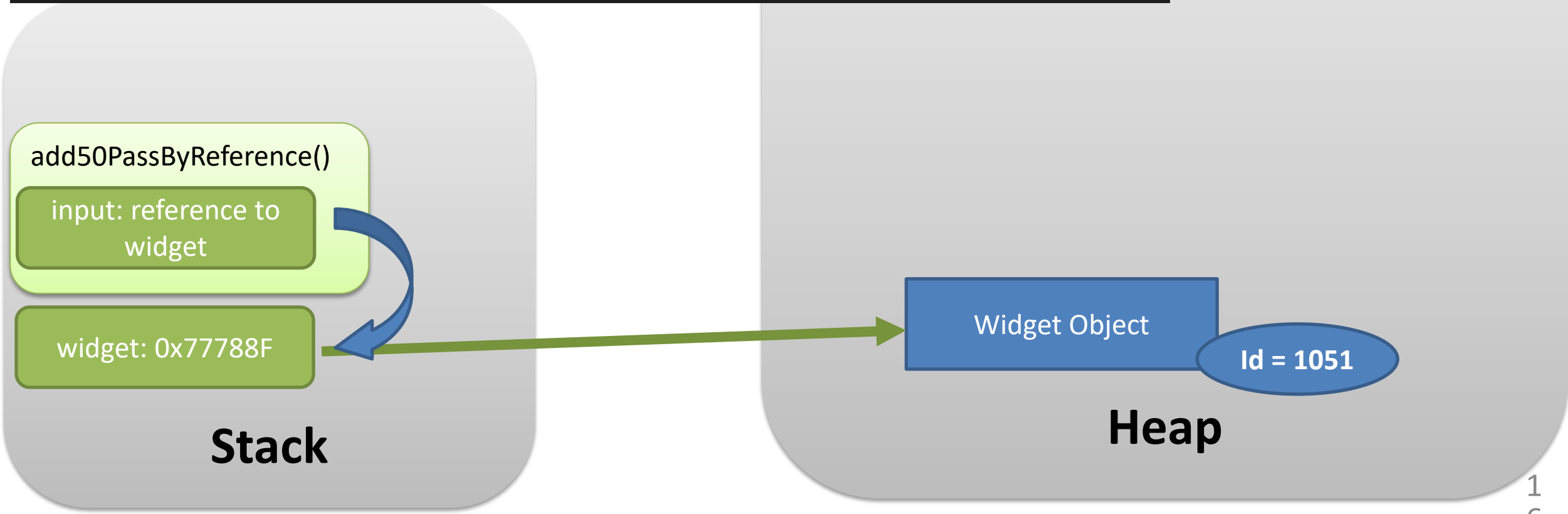
Reference Type – Pass by Reference

```
1 reference
private static void add50PassByReference(ref Widget input)
{
    input.Id += 50;
    Console.WriteLine($"{labelInside,-36} {input.Id,6:N0}");
}
```



Reference Type – Pass by Reference

```
1 reference
private static void add50PassByReference(ref Widget input)
{
    input.Id += 50;
    Console.WriteLine($"{labelInside,-36} {input.Id,6:N0}");
}
```



Reference Type – Pass by Value

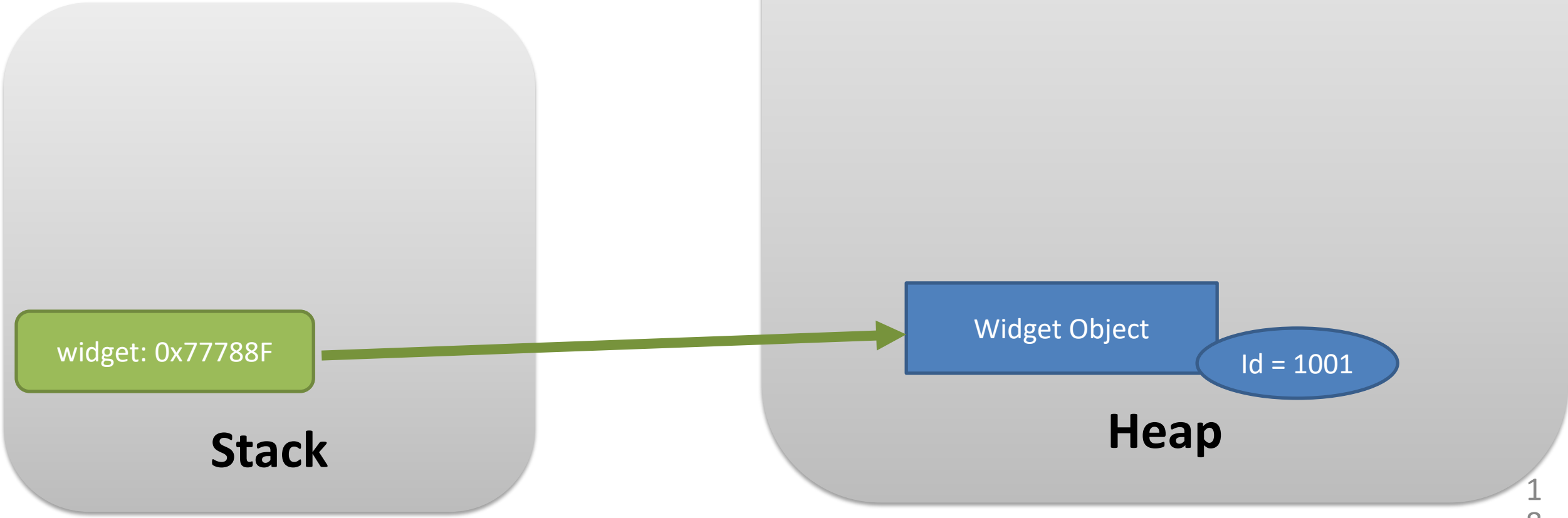
```
0 references
static void Main(string[] args)
{
    Console.Title = "Reference Type Pass By Value Example";

    Widget widget = new Widget { Id = 1001 };

    Console.WriteLine($"{labelOriginal, -36} {widget.Id, 6:N0}");

    add50PassByValue(widget);

    Console.WriteLine($"{labelAfter, -36} {widget.Id ,6:N0}\n\n");
}

1 reference
private static void add50PassByValue(Widget input)
{
    input.Id += 50;
    Console.WriteLine($"{labelInside, -36} {input.Id, 6:N0}");
}
```

```
Reference Type Pass By Value Example
Original value:                          1,001
Value inside PassByValue method:         1,051
Value after PassByValueCall:             1,051
```

- A copy of the reference is passed to the method.
- Copied reference still refers to the original object.
- Any actions taken in the method permanently change the object.
- **Changing the reference itself will not be permanent, as the parameter passed itself cannot be changed**

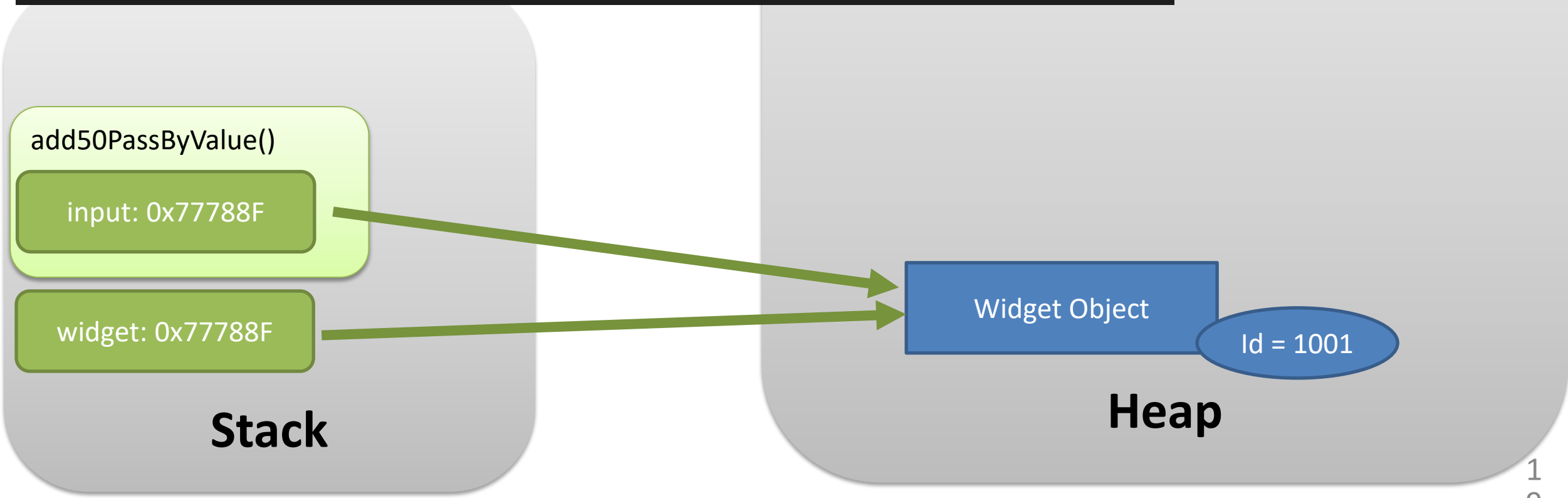
Reference Type – Pass by Value

```
Console.Title = "Reference Type Pass By Value Example";

Widget widget = new Widget { Id = 1001 };

Console.WriteLine($"{labelOriginal,-36} {widget.Id,6:N0}");
```



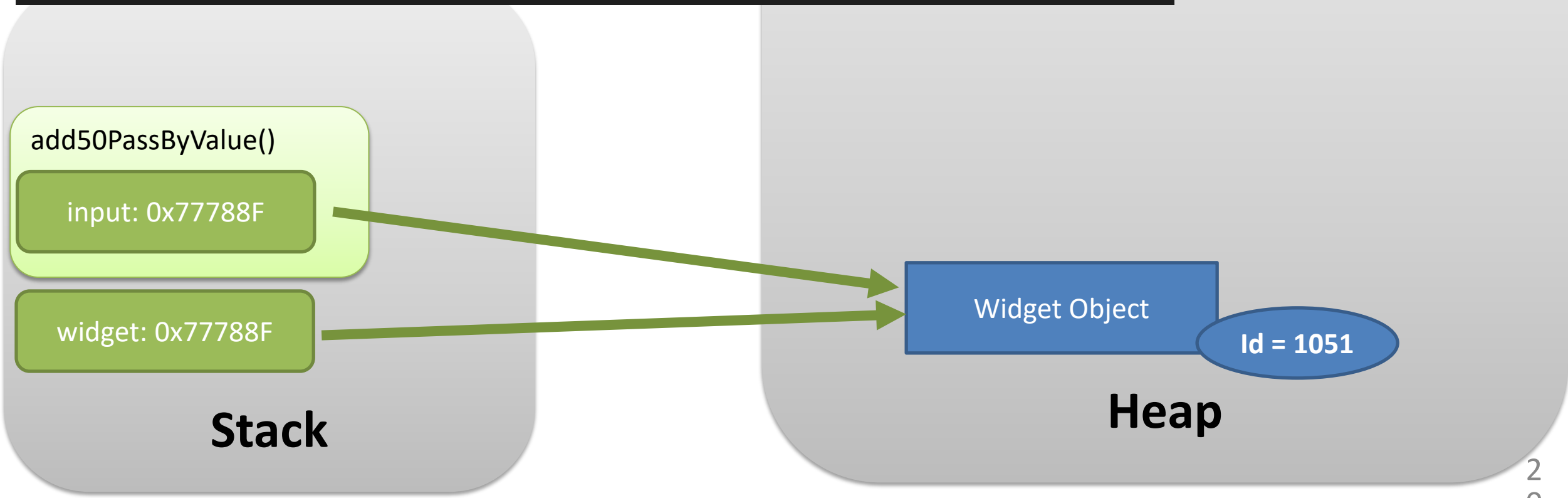
Reference Type – Pass by Value

```
1 reference
private static void add50PassByValue(Widget input)
{
    input.Id += 50;
    Console.WriteLine($"{labelInside,-36} {input.Id,6:N0}");
}
```



Reference Type – Pass by Value

```
1 reference
private static void add50PassByValue(Widget input)
{
    input.Id += 50;
    Console.WriteLine($"{labelInside,-36} {input.Id,6:N0}");
}
```

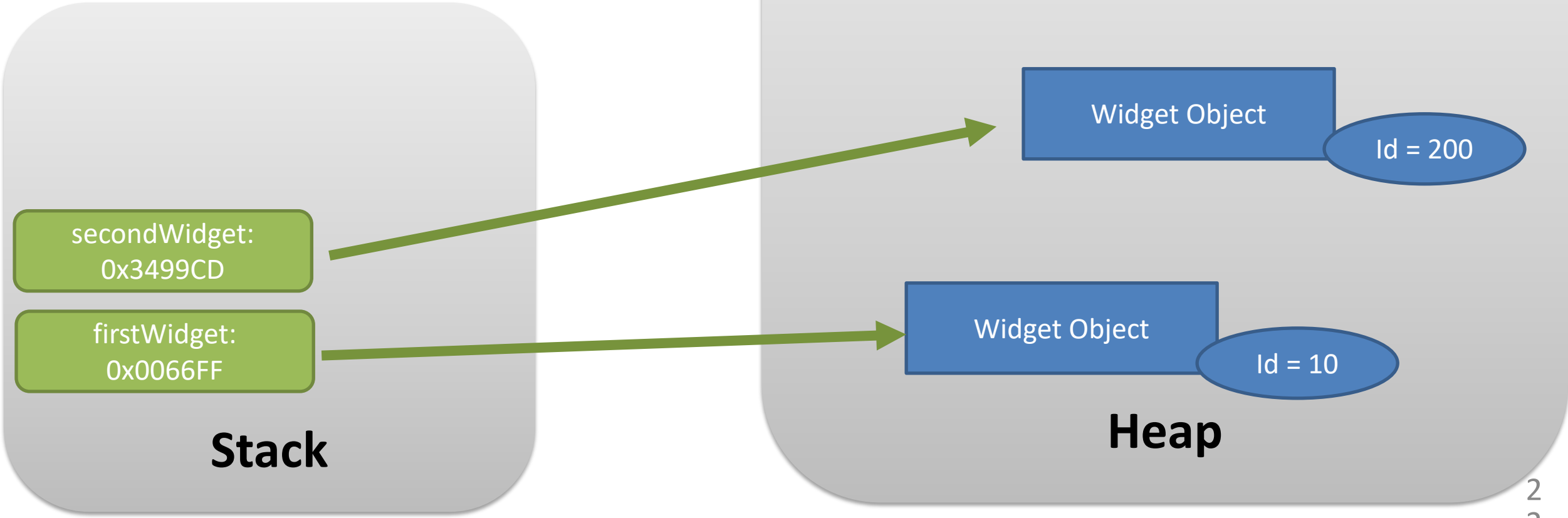


Reference Type – Reassignment

```
1 reference
static void SwapWidgetByValue(Widget widgetOne, Widget widgetTwo)
{
    widgetOne = widgetTwo;
}

1 reference
static void SwapWidgetByRef(ref Widget widgetOne, ref Widget widgetTwo)
{
    widgetOne = widgetTwo;
}
```

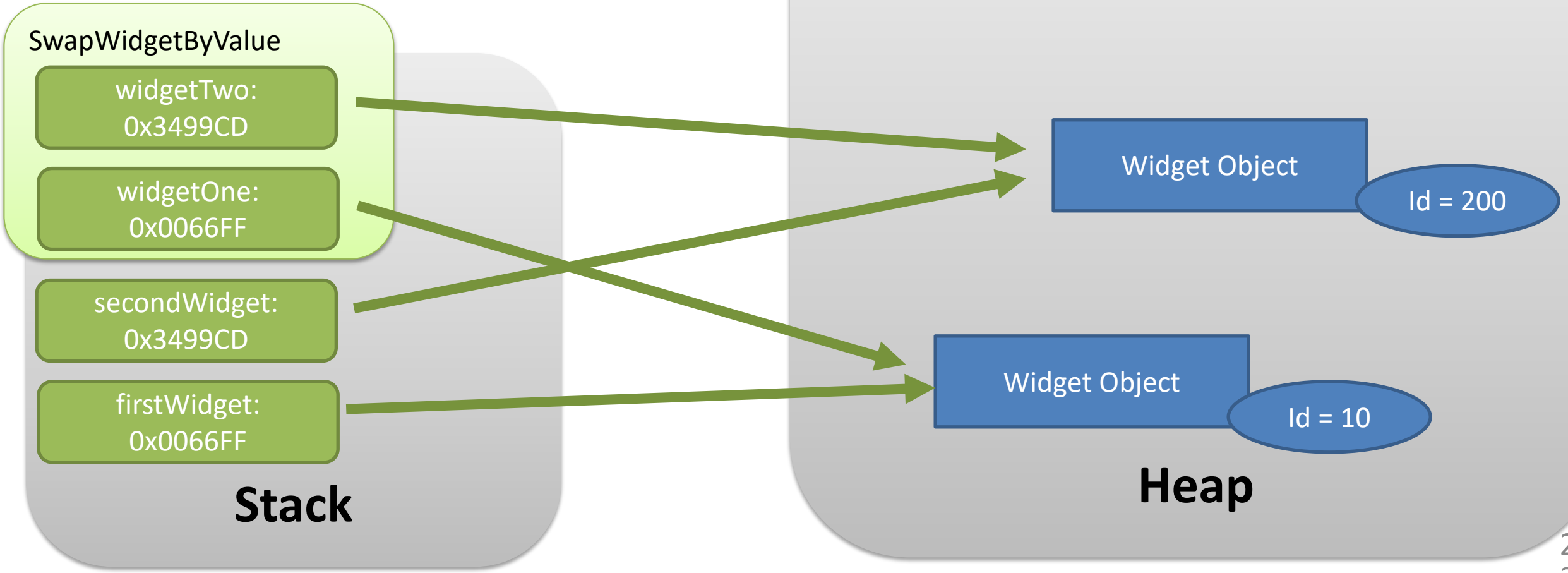
Reference Type – Reassignment – By Value

```
Widget firstWidget = new Widget();
Widget secondWidget = new Widget();

firstWidget.Id = 10;
secondWidget.Id = 200;
```



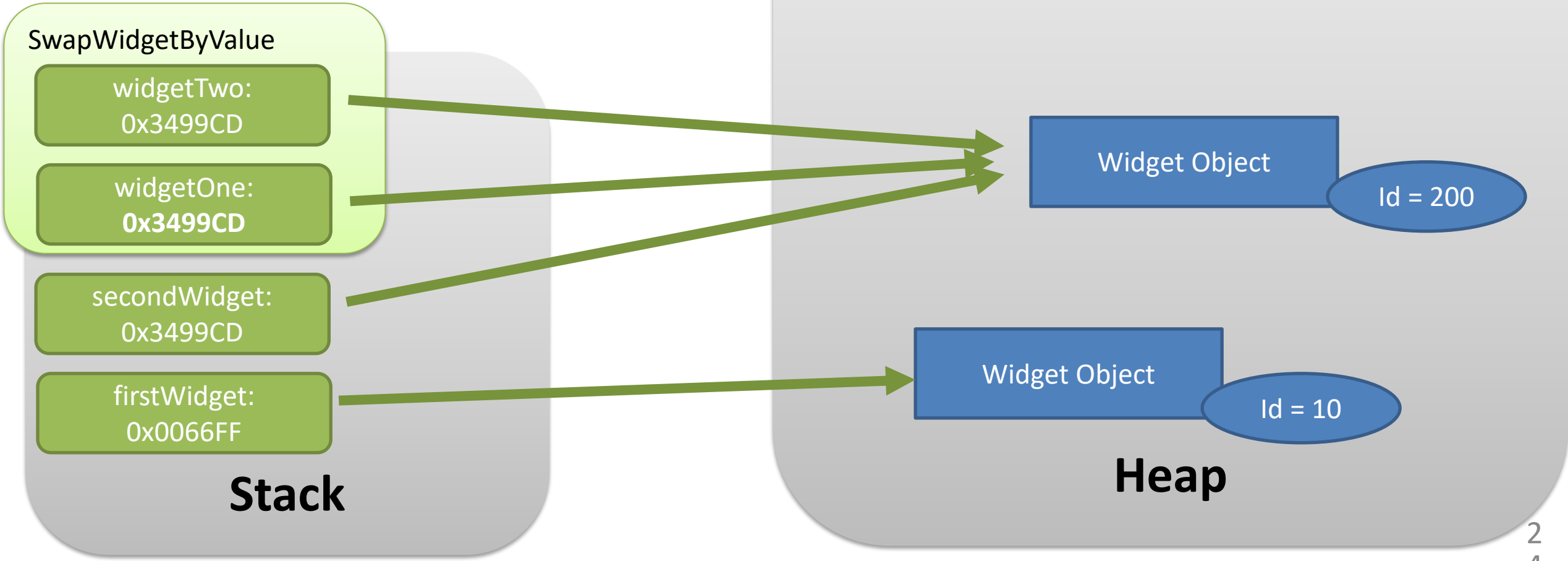
Reference Type – Reassignment – By Value

```
1 reference
static void SwapWidgetByValue(Widget widgetOne, Widget widgetTwo)
{
    widgetOne = widgetTwo;
}
```



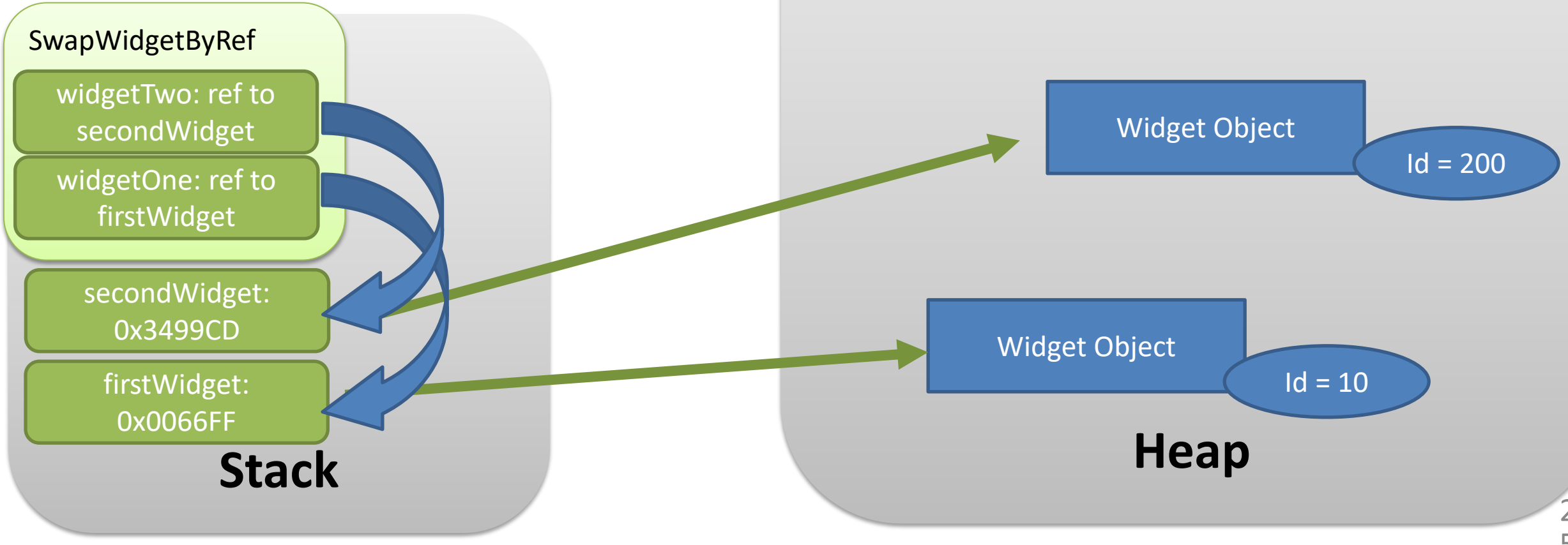
Reference Type – Reassignment – By Value

```
1 reference
static void SwapWidgetByValue(Widget widgetOne, Widget widgetTwo)
{
    widgetOne = widgetTwo;
}
```



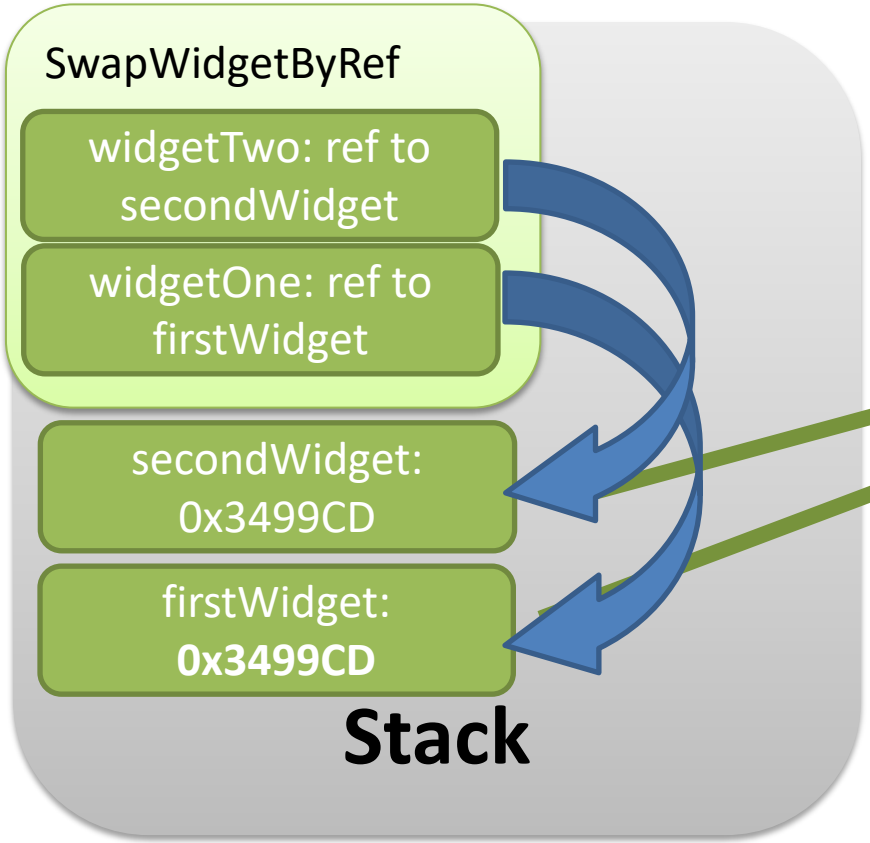
Reference Type – Reassignment – By Reference

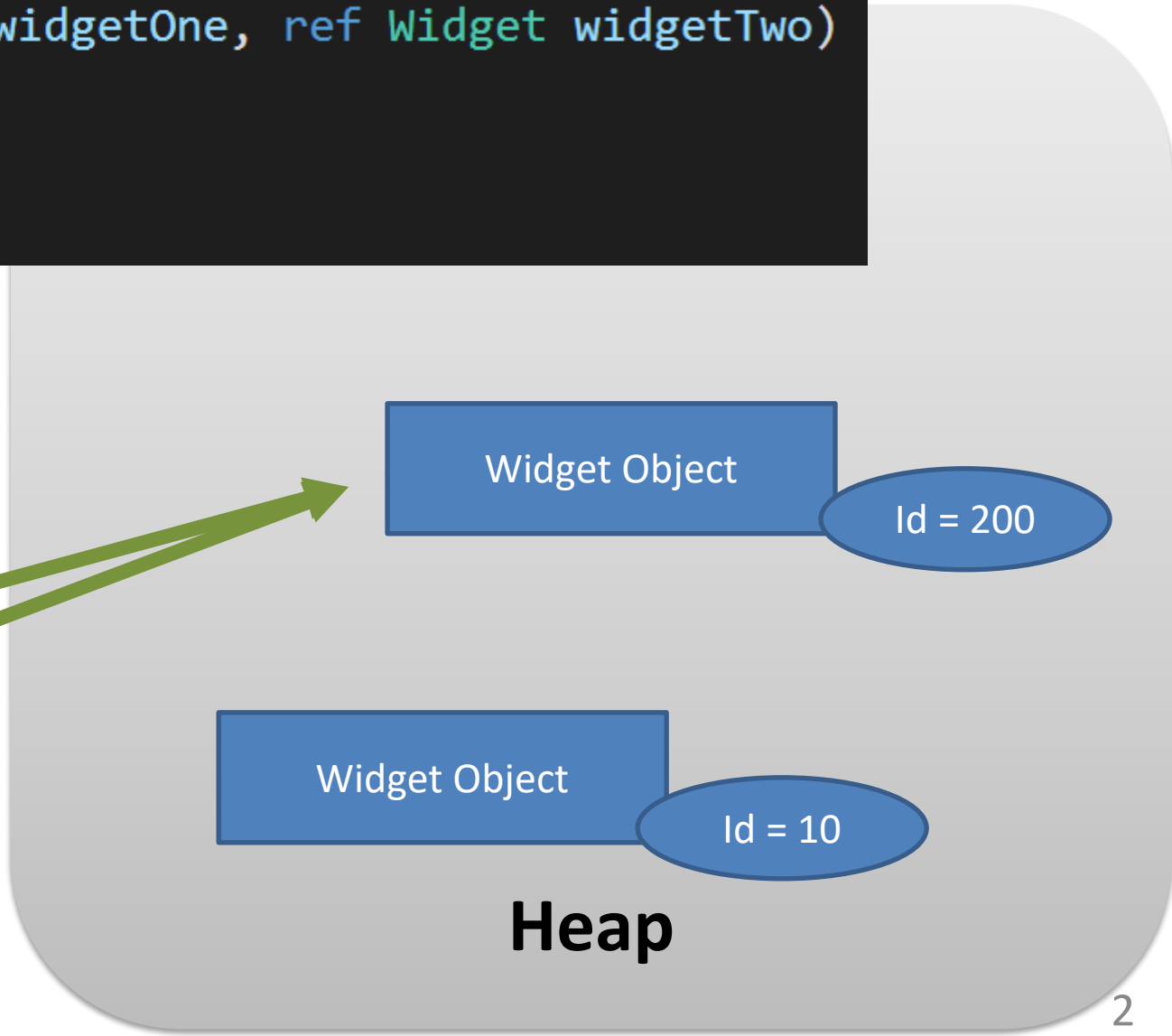
```
1 reference
static void SwapWidgetByRef(ref Widget widgetOne, ref Widget widgetTwo)
{
    widgetOne = widgetTwo;
}
```



Reference Type – Reassignment – By Reference

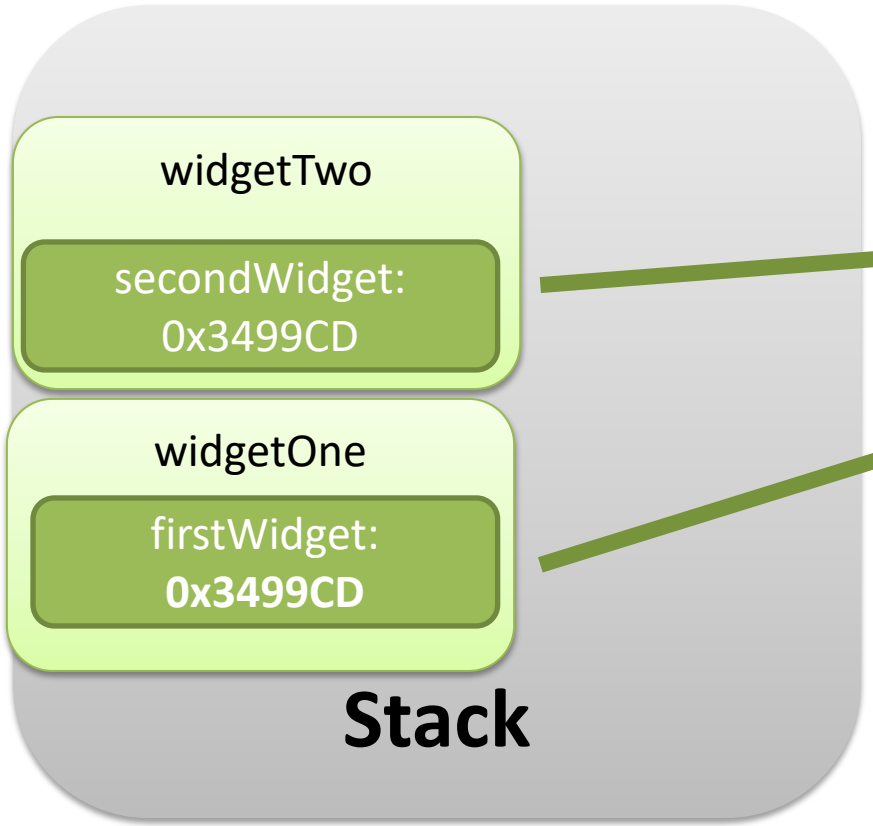
```
1 reference
static void SwapWidgetByRef(ref Widget widgetOne, ref Widget widgetTwo)
{
    widgetOne = widgetTwo;
}
```

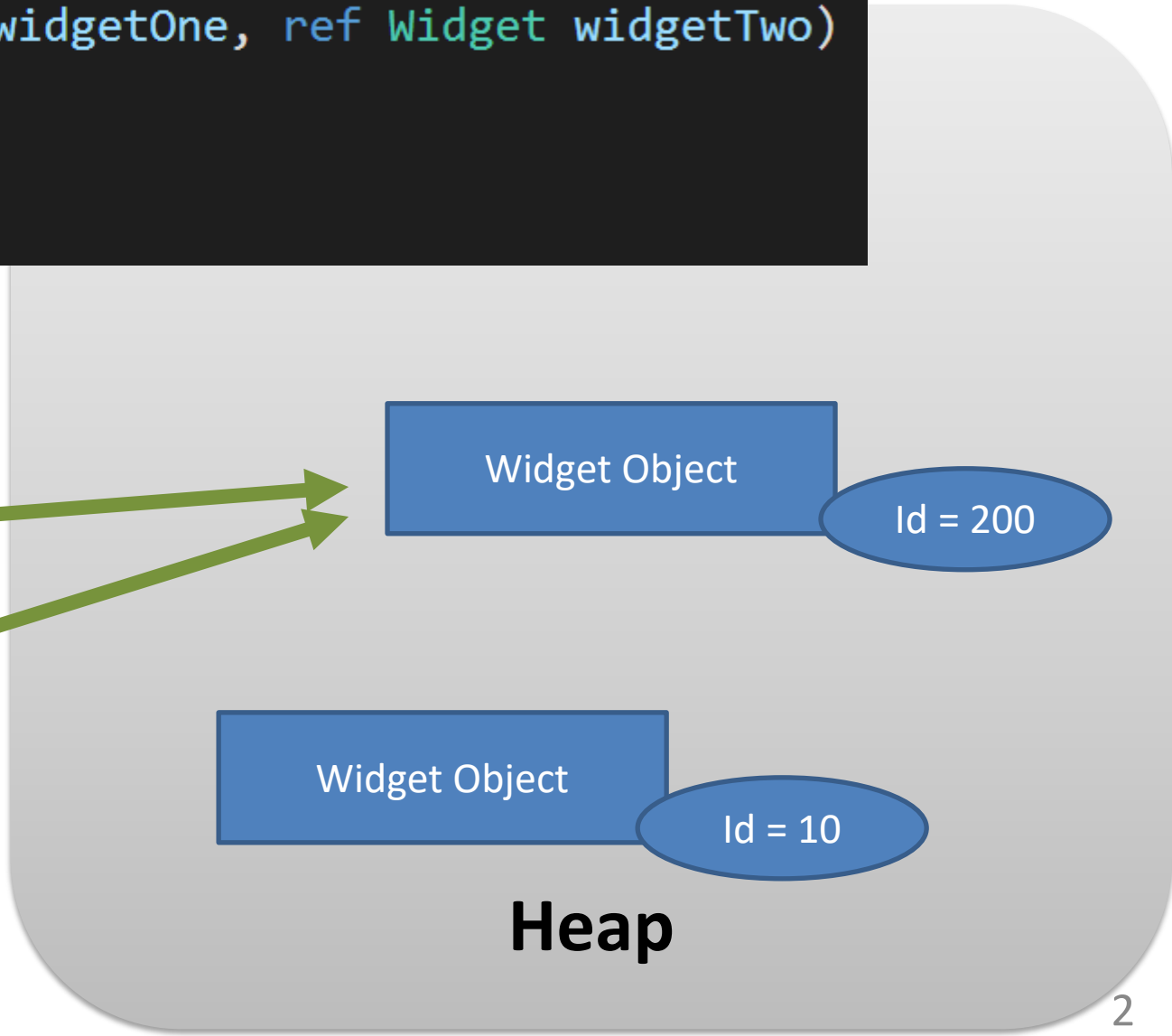




Reference Type – Reassignment – By Reference

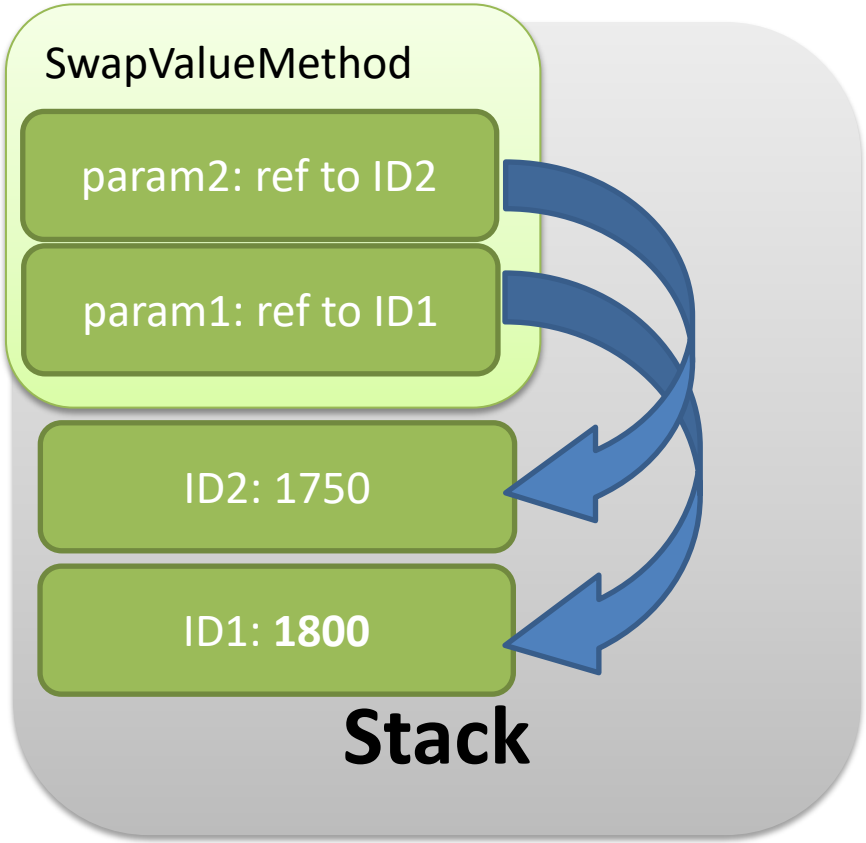
```
1 reference
static void SwapWidgetByRef(ref Widget widgetOne, ref Widget widgetTwo)
{
    widgetOne = widgetTwo;
}
```

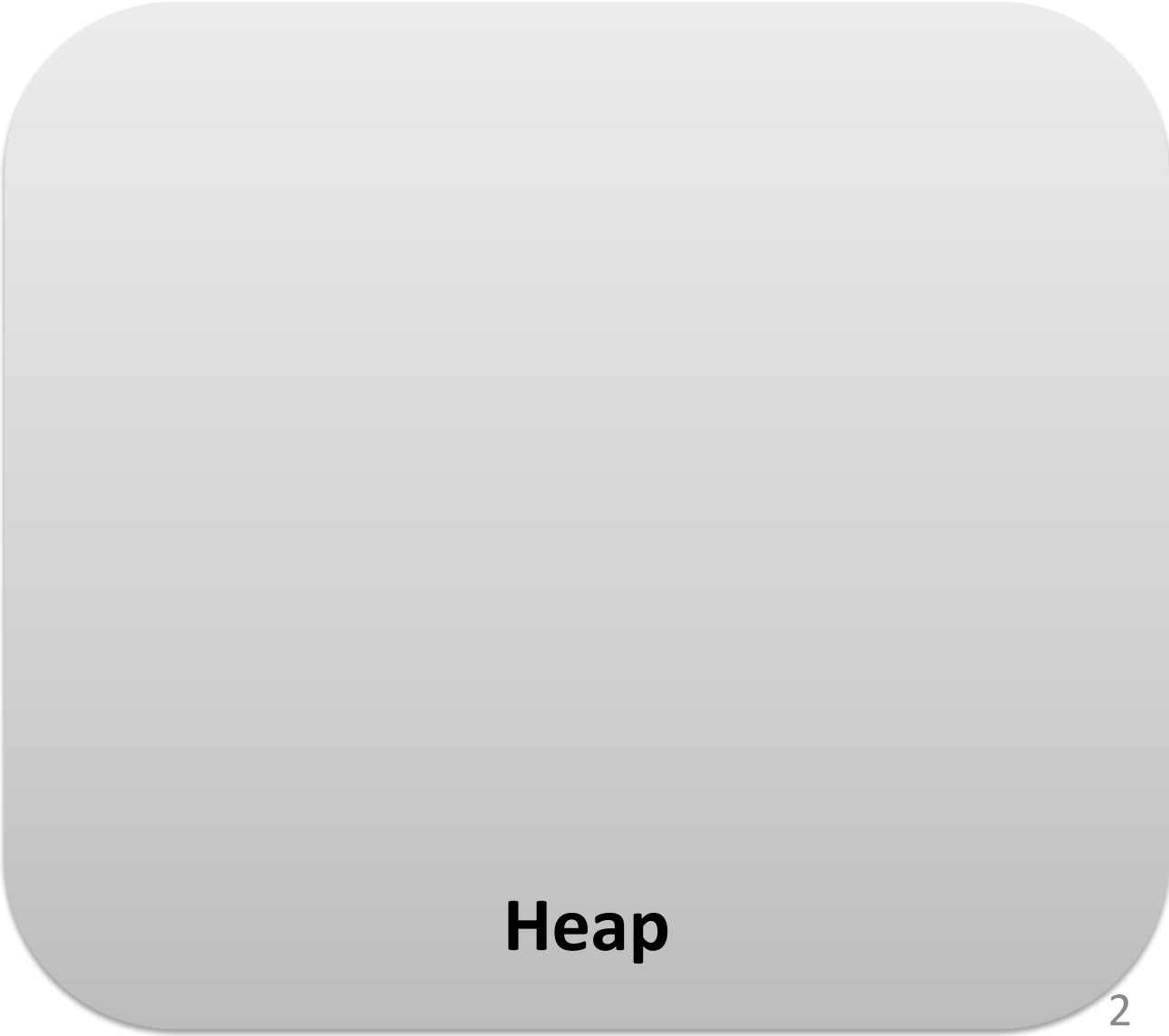




Value Type – Reassignment – By Reference

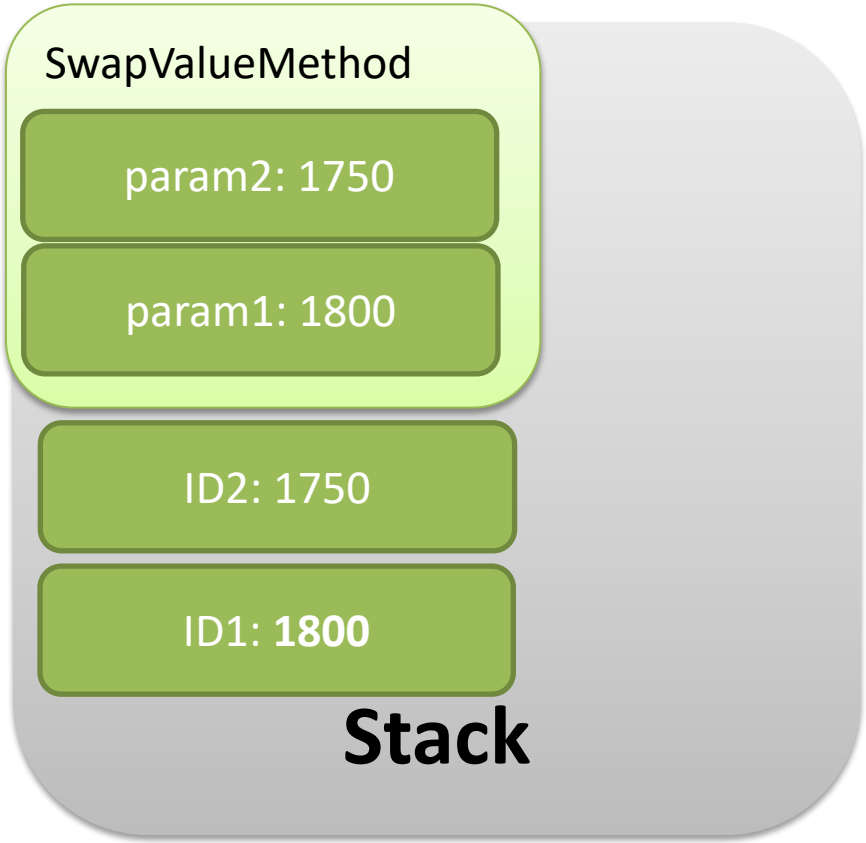
```
private void swapValues(ref int param1, ref int param2)
{
    param2=param1;
}
//outcome: permanent change
```

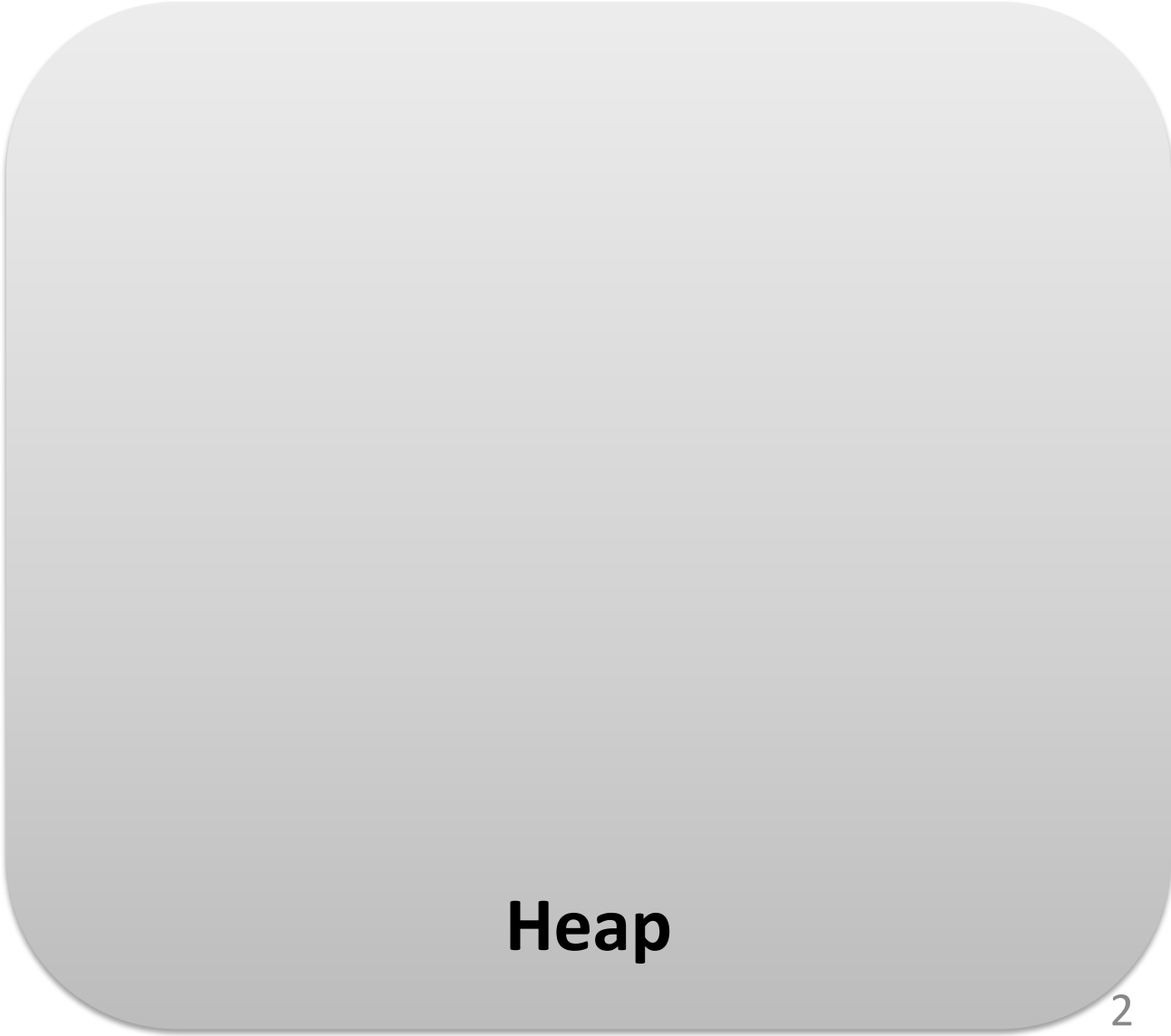




Value Type – Reassignment – By Value

```
private void swapValues(int param1, int param2)
{
    param2=param1;
}
//Outcome: original values not changed
```





Output Parameters

```
23
24          processArray(out minimum
25                     , out maximum
26                     , out count
27                     , out sum
28                     , out average
29                     , 5, 2, 9, 23, 15, 28, 7, 11);
30      }
31
        1 reference
32      private static void processArray(out int minimum
33                                     , out int maximum
34                                     , out int count
35                                     , out int sum
36                                     , out double average
37                                     , params int[] input)
38      {
39          sum = 0;
40
```

- Forces initialization inside the method
- **Does not require initialization before calling the method**
- Passes by-reference
- Must use the out keyword in the method definition and at the call site
- Can be used with other parameter types in the same method

Parameter Initialization Requirements

Parameter Type	Initialization
Default (by-value)	Mandatory
Ref (by-reference)	Mandatory
Out (by-reference)	Optional

Ref vs Out

Out requires that the function initializes the object.

Ref does not force the object to be initialized in the function.

So, **ref** implies that the object exists already and is being modified, while **out** implies that the object is being set inside the function in order to be returned → eg, it's a way to set many variables at once inside one function.

Params Parameters

```
23
24          processArray(out minimum
25                     , out maximum
26                     , out count
27                     , out sum
28                     , out average
29                     , 5, 2, 9, 23, 15, 28, 7, 11);
30      }
31
        1 reference
32      private static void processArray(out int minimum
33                                     , out int maximum
34                                     , out int count
35                                     , out int sum
36                                     , out double average
37                                     , params int[] input)
38      {
39          sum = 0;
40
```

Discrete int parameters are passed to the method as an array

params must be the last parameter in the signature

- Accepts multiple parameters of the same type
- Compiler automatically converts passed parameters into an array (elements are passed by value)
- params keyword required in method definition only
- Can be used with other parameters val/ref/out
- Must be the last parameter in the signature

Optional Parameters

```
12      static void Main(string[] args)
13      {
14          Console.WriteLine("Total: {0, 6:N0}", sumIntegers(100, 200, 300));
15          Console.WriteLine("Total: {0, 6:N0}", sumIntegers(100, 200));
16          Console.WriteLine("Total: {0, 6:N0}\n", sumIntegers(100));
17          // Can not "skip" an optional parameter
18          //Console.WriteLine("Total: {0, 6:N0}", sumIntegers(100,,300)); // no go
19
20          Console.WriteLine("Total: {0, 6:N0}", sumIntegersOld(100, 200, 300));
21          Console.WriteLine("Total: {0, 6:N0}", sumIntegersOld(100, 200));
22          Console.WriteLine("Total: {0, 6:N0}\n", sumIntegersOld(100));
23
24          Console.WriteLine("Value: {0, 6:N0}", getValue());
25          Console.WriteLine("Value: {0, 6:N0}\n", getValue(335));
26      }
27
28      // all parameters can be optional
        2 references
29      private static int getValue(int value = 50)
30      {
31          return value;
32      }
33
34      // optional parameters must occur last in the signature
        3 references
35      private static int sumIntegers(int first, int second = 20, int third = 30)
36      {
37          return first + second + third;
38      }
39
```

```
Optional Parameters Example
Total:     600
Total:     330
Total:     150

Total:     600
Total:     330
Total:     150

Value:      50
Value:     335
```

- Provides default values for parameters
- Optional parameters must come after required parameters
- All parameters may be optional

Optional Parameters

```
42      // previous approach to solve this
        2 references
43      private static int sumIntegersOld(int first, int second, int third)
44      {
45          return first + second + third;
46      }
47
        2 references
48      private static int sumIntegersOld(int first, int second)
49      {
50          return sumIntegersOld(first, second, 30); // calls 3 argument overload
51      }
52
        1 reference
53      private static int sumIntegersOld(int first)
54      {
55          return sumIntegersOld(first, 20); // calls 2 argument overload
56      }
57
```

- All optional parameters passed by val (no ref / out)
- Required parameters can be passed by val / ref / out

Named Parameters

```
12      static void Main(string[] args)
13      {
14          // standard method call
15          displayPerson("Moe", "Howard", 63);
16
17          // call with parameter naming
18          displayPerson(firstName: "Larry", lastName: "Fine", age: 61);
19
20          // call with parameter naming - alternate order
21          displayPerson(age: 58, lastName: "Howard", firstName: "Curley");
22
23          // works with constructors as well
24          Widget widget = new Widget(id: 1001
25                                     , price: 2200m
26                                     , description: "Really nice widget");
27
28          Console.WriteLine("ID:    {0}\nDesc:  {1}\nPrice: {2:N2}\n"
29                  , widget.Id
30                  , widget.Description
31                  , widget.Price);
32      }
33
34      // method defined as usual
        3 references
35      private static void displayPerson(string firstName, string lastName, int age)
36      {
37          Console.WriteLine("Name: {0} {1}\nAge:  {2}\n", firstName, lastName, age);
38      }
39
40
```

- Improves code readability (particularly when method signatures are long)
- No changes required in method definition (syntax changes at call site only)
- Can alter parameter order
- Call is in the form: `paramName: value`
- Can be used with ref/out: `paramName: ref value`

Casting

```
short value = 1076;
int amount = value;
```

Use standard “C Style” casting:

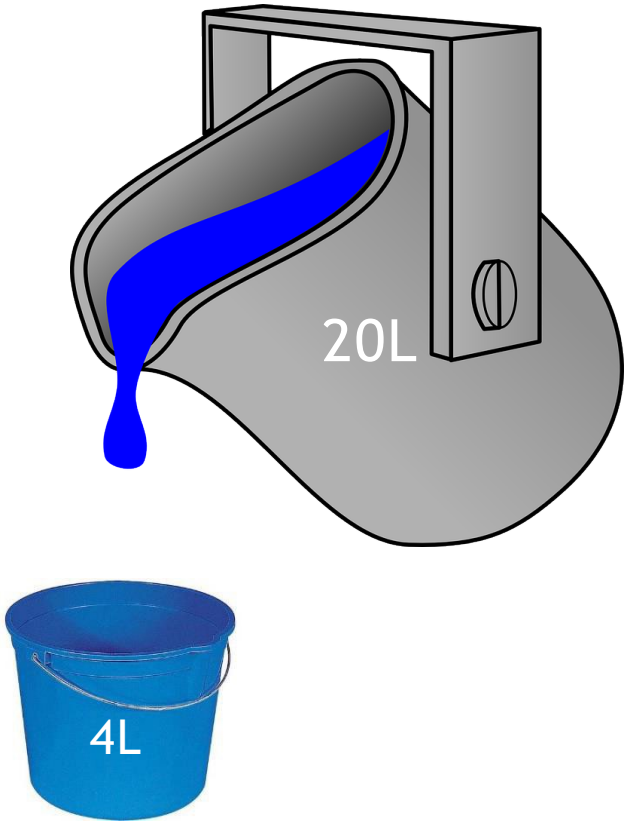
```
int value = 1076;
short amount = (short)value;
```

Some conversions do not require a cast (implicit)	Some conversions require casting (explicit)
short to int	int to short
int to long	long to int
int to float	float to int (truncates)
float to double	double to float
Customer to Object	Object to Customer

Casting

short to **int**
(implicit)

int to **short**
(explicit)



Casting

long to **int** (explicit) – extra bytes are simply cut off

```
0 references
static void Main(string[] args)
{

    long inputLong = 99999999990;
    int inputInt = (int)inputLong;
    Console.WriteLine($"Before: {inputLong}, after:{inputInt}");
}
```

Microsoft Visual Studio Debug Console

```
Before: 99999999990, after:1215752182
```

Casting

long to int (explicit) – extra bytes are simply cut off

long variables have **8** bytes, **int** have **4** bytes

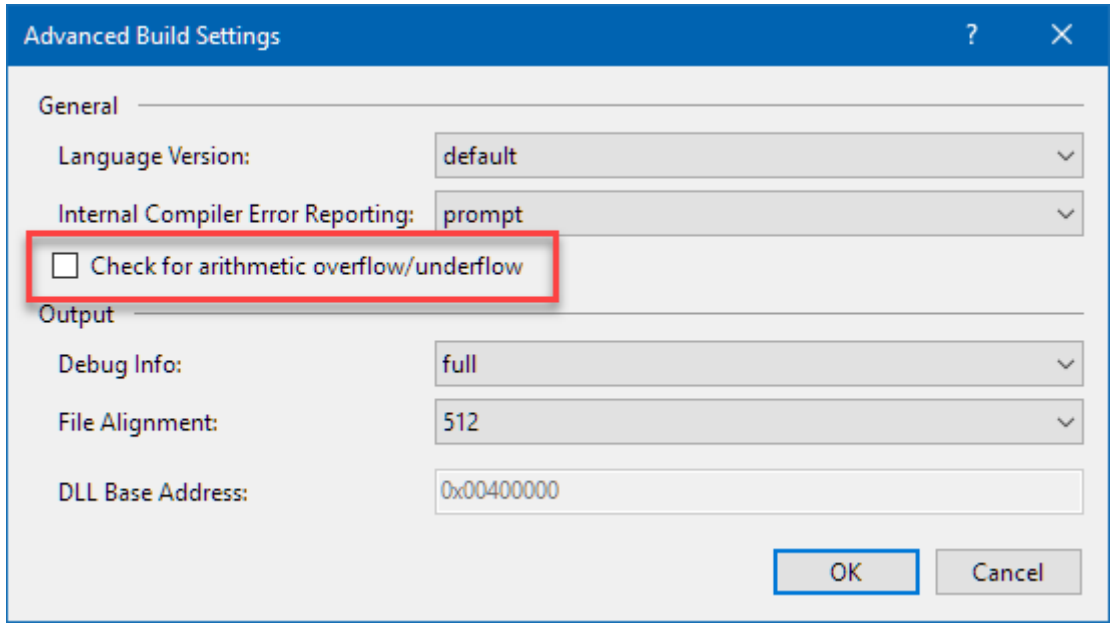
99999999990 in *binary* is:
1011101001000011101101110011111110110

1215752182 in *binary* is:
1001000011101101110011111110110

	Byte 8	Byte 7	Byte 6	Byte 5	Byte 4	Byte 3	Byte 2	Byte 1
long	00000000	00000000	00000000	000**10111**	**01001000**	**01110110**	**11100111**	**11110110**
int					01001000	01110110	11100111	11110110

checked / unchecked

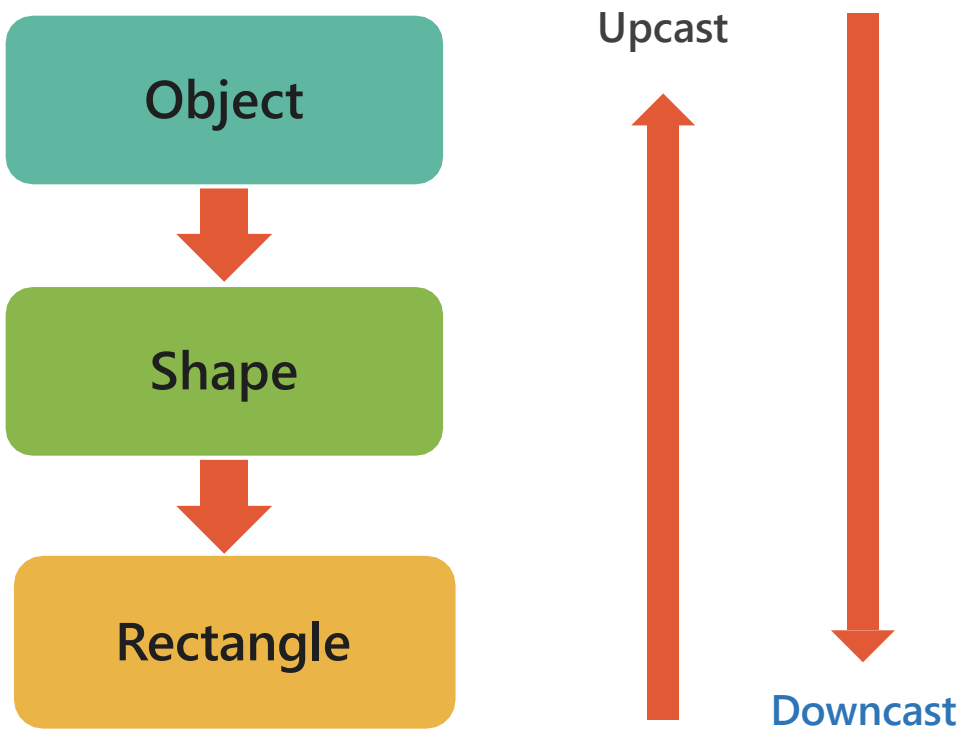
Click Project ► Properties




```
21
22      checked
23      {
24          shortVal = 20000;
25          Console.WriteLine($"Value is: {shortVal:N0}");
26
27          shortVal += adder;
28          Console.WriteLine($"Value is: {shortVal:N0}");
29      }
30
```

- Turns overflow and underflow checking on/off at runtime. (default is off)
- Can alter global setting by placing suspect code in a marked code block.
- Always ensure that overflow and underflow checking is turned off for a production build.

Object Upcasting and Downcasting



- **"Is a"** Relationship
- Every Rectangle is a Shape
- Not every Shape is a Rectangle
- Upcast is implicit:
  Shape = Rectangle
- Downcast is explicit:
  Rectangle = (Rectangle) Shape

Convert.To___

```
14      string input;
15      int age;
16
17      Console.Write("Enter your age in years: ");
18      input = Console.ReadLine();
19
20      age = Convert.ToInt32(input);
21
```

```
19          age = Convert.To
20
21
22
23
24
25
26
27
```

- ToBase64CharArray
- ToBase64String
- ToBoolean
- ToByte
- ToChar
- ToDateTime
- ToDecimal
- ToDouble
- ToInt16

- The Convert class contains several static methods to convert an input value to the method's associated type.
- An Exception is thrown if the conversion fails.
- The method names map to the .NET Type names and not the C# aliases.

Parse Static Method

```
13
14      Console.Write("Enter your age in years: ");
15      string input = Console.ReadLine();
16
17      int age = int.Parse(input);
18
19      Console.WriteLine($"You entered: {age}");
20
```

Static method to convert a string value to a numeric, DateTime or boolean value:

```
int.Parse
float.Parse
double.Parse
bool.Parse
DateTime.Parse
others...
```

TryParse Static Method

First Parameter (string)
String to be converted

```
14
15      int age;
16
17      Console.WriteLine("Enter your age in years: ");
18      string ageInput = Console.ReadLine();
19
20      bool success = int.TryParse(ageInput, out age);
21
```

Returns bool
Success true
Failure false

Second Parameter (out int)
Success value represented by input
Failure zero

A safer means of converting a string value to a numeric, DateTime or boolean value:

```
int.TryParse
float.TryParse
double.TryParse
bool.TryParse
DateTime.TryParse
others...
```

Parsing Enums

Can also use TryParse() or Parse() with Enums

```
string stringDay = "Thursday";
if(Enum.TryParse(stringDay, out DayOfWeek parsedEnumDay))
{
    Console.WriteLine($"The dat passed in as a string parsed to this enum: {parsedEnumDay}");
    if(parsedEnumDay == DayOfWeek.Sunday)
    {
        Console.WriteLine("The day passed in as a string was Sunday!");
    } else
    {
        Console.WriteLine("The day passed in as a string was NOT Sunday!");
    }
}
```

Casting/Converting/Parsing Summary

- **Casting**
  - Attempts to *change* the type of the value
- **Converting**
  - Attempts to *convert* the data to another type and may return a modified value in helpful ways (eg rounding for float to int)
- **Parsing**
  - Attempts to convert *string* data to another type

Exception Handling

```
15      try
16      {
17          int age = Convert.ToInt32("10");
18          Console.WriteLine("Next statement in Try Block");
19      }
20
21      catch (IndexOutOfRangeException ex)
22      {
23          Console.WriteLine("Handling type IndexOutOfRange");
24      }
25
26      catch (FormatException ex)
27      {
28          Console.WriteLine("Handling type FormatException");
29      }
30
31      catch (Exception ex)
32      {
33          Console.WriteLine("Handling type Exception");
34      }
35
36      finally
37      {
38          Console.WriteLine("Finally Block executed");
39      }
40
41      Console.WriteLine("Next statement in program");
42
```

Try Block
Place potentially problematic code here

Catch Block (specific)
Handles a specific type of exception

Catch Block (specific)
Handles a specific type of exception

Catch Block (general)
Handles any type of exception (must be last)

Finally Block (optional)
Will always execute. Good for resource cleanup

Exception Handling

```
15      try
16      {
17          int age = Convert.ToInt32("10");
18          Console.WriteLine("Next statement in Try Block");
19      }
20
21      catch (IndexOutOfRangeException ex)
22      {
23          Console.WriteLine("Handling type IndexOutOfRange");
24      }
25
26      catch (FormatException ex)
27      {
28          Console.WriteLine("Handling type FormatException");
29      }
30
31      catch (Exception ex)
32      {
33          Console.WriteLine("Handling type Exception");
34      }
35
36      finally
37      {
38          Console.WriteLine("Finally Block executed");
39      }
40
41      Console.WriteLine("Next statement in program");
42
```

Normal Execution

Exception Handling Demo

```
Next statement in Try Block
Finally Block executed
Next statement in program
```

- No Exceptions thrown
- Execution passed to Finally Block
- Entire Try Block executed
- Execution continues following Finally Block

Exception Handling

```
15      try
16      {
17          int age = Convert.ToInt32("ten");
18          Console.WriteLine("Next statement in Try Block");
19      }
20
21      catch (IndexOutOfRangeException ex)
22      {
23          Console.WriteLine("Handling type IndexOutOfRange");
24      }
25
26      catch (FormatException ex)
27      {
28          Console.WriteLine("Handling type FormatException");
29      }
30
31      catch (Exception ex)
32      {
33          Console.WriteLine("Handling type Exception");
34      }
35
36      finally
37      {
38          Console.WriteLine("Finally Block executed");
39      }
40
41      Console.WriteLine("Next statement in program");
42
```

Exception Thrown

```
Exception Handling Demo
Handling type FormatException
Finally Block executed
Next statement in program
```

- Exception thrown in Try Block (line 17)
- Execution passed to specific Catch Block
- Catch Block executed
- Execution passed to Finally Block
- Execution continues following Finally Block

Exception Handling

```
15      try
16      {
17          int age = Convert.ToInt32("ten");
18          Console.WriteLine("Next statement in Try Block");
19      }
20
21      catch (IndexOutOfRangeException ex)
22      {
23          Console.WriteLine("Handling type IndexOutOfRange");
24      }
25
26      //catch (FormatException ex)
27      //{
28      //    Console.WriteLine("Handling type FormatException");
29      //}
30
31      catch (Exception ex)
32      {
33          Console.WriteLine("Handling type Exception");
34      }
35
36      finally
37      {
38          Console.WriteLine("Finally Block executed");
39      }
40
41      Console.WriteLine("Next statement in program");
42
```

Exception Thrown

Exception Handling Demo

```
Handling type Exception
Finally Block executed
Next statement in program
```

- Exception thrown in Try Block (line 17)
- Execution passed to general Catch Block (No specific matching Block)
- Catch Block executed
- Execution passed to Finally Block
- Execution continues following Finally Block

Reading Text Files

```
18      System.IO.StreamReader streamReader = null; // or using System.IO
19      string path = @"..\..\sample.txt";
20      string lineData;
21
22      if (!File.Exists(path))
23      {
24          Console.WriteLine("\nFile not found.\n");
25      }
26      else
27      {
28          try
29          {
30              streamReader = new StreamReader(path);
31
32              while ((lineData = streamReader.ReadLine()) != null)
33              {
34                  Console.WriteLine(lineData);
35              }
36          }
37
38          catch (Exception ex)
39          {
40              Console.WriteLine($"\n{ex.Message}\n");
41          }
42
43          finally
44          {
45              if (streamReader != null)
46              {
47                  streamReader.Close();
48              }
49          }
50      }
51
```

StreamReader (path)
Opens the text file specified by the path

ReadLine() method

Reads to the next NewLine character and returns the data as string (Strips NewLine)

[Returns null if no further data]

File Pointer advances to the position immediately following NewLine)

Null Test
Calling Close() on a null StreamReader object will throw an exception (Not Good!!)

Close() method
Always call Close() to close the file

Reading Text Files (with using)

```
18      System.IO.StreamReader streamReader = null; // or using System.IO;
19      string path = @"..\..\sample.txt";
20      string lineData;
21
22      if (!File.Exists(path))
23      {
24          Console.WriteLine("\nFile not found.\n");
25      }
26      else
27      {
28          try
29          {
30              using (streamReader = new StreamReader(path))
31              {
32                  while ((lineData = streamReader.ReadLine()) != null)
33                  {
34                      Console.WriteLine(lineData);
35                  }
36              }
37          }
38
39          catch (Exception ex)
40          {
41              Console.WriteLine($"\n{ex.Message}\n");
42          }
43      }
44
```

Using Block

Placing the instantiation of the StreamReader in a using block will automatically close the file and clean up the instance when execution in the block completes

StreamReader (path)

Opens the text file specified by the path

No Finally Block

A finally block is not required as the using block handles file closing and resource cleanup

Reading Text Files (with using declaration - C# 8.0)

C# Copy

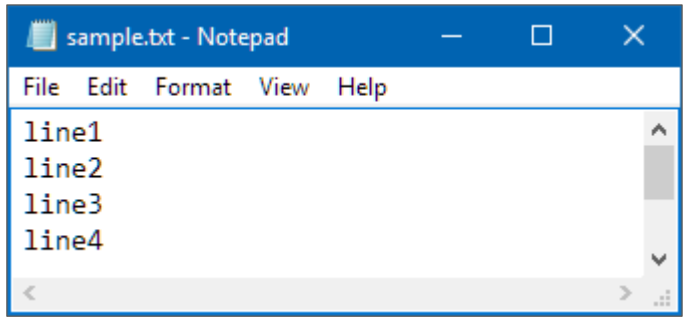
```
if (...)
{
   using FileStream f = new FileStream(@"C:\users\jaredpar\using.md");
   // statements
}

// Equivalent to
if (...)
{
   using (FileStream f = new FileStream(@"C:\users\jaredpar\using.md"))
   {
    // statements
   }
}
```

The lifetime of a `using` local will extend to the end of the scope in which it is declared. The `using` locals will then be disposed in the reverse order in which they are declared.

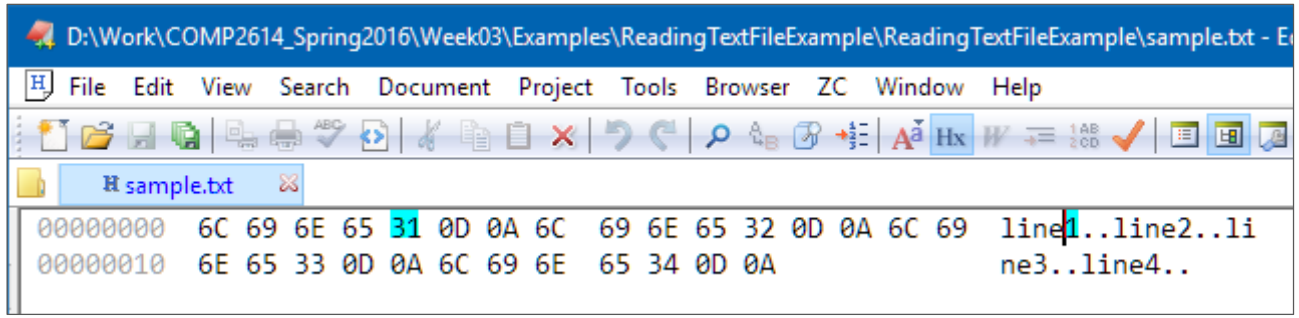
From: *https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-8.0/using*

Reading Text Files

sample.txt open in Notepad

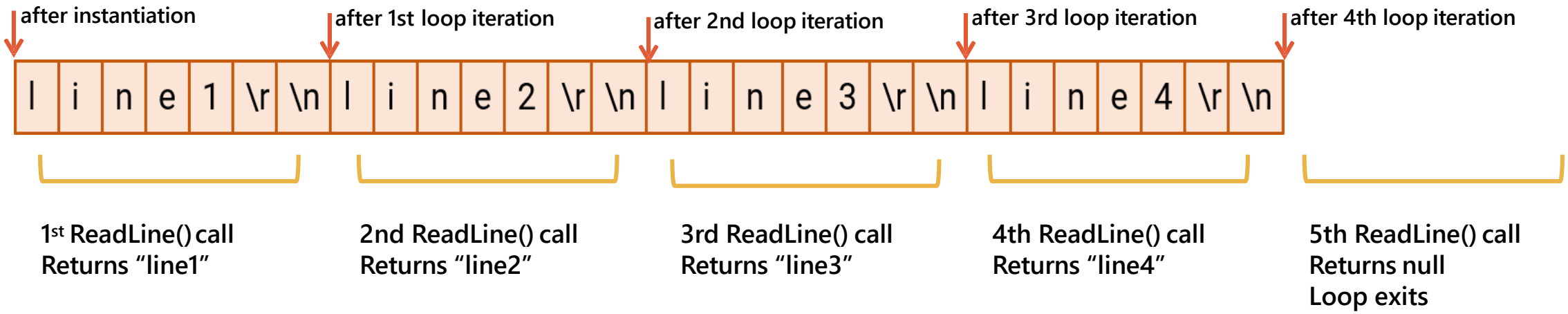


sample.txt open in the EditPlus hex editor




```
// StreamReader instantiation opens file and places file pointer at beginning of file
StreamReader streamReader = new StreamReader("..\..\sample.txt");
```

File Pointer position



Command Line Arguments

```
0 references
12  static void Main(string[] args)
13  {
14      if (args.Length > 0)
15      {
16          for (int index = 0; index < args.Length; index++)
17          {
18              Console.WriteLine("args[{0}] == {1}", index, args[index]);
19          }
20      }
21  }
22
```

Command Prompt

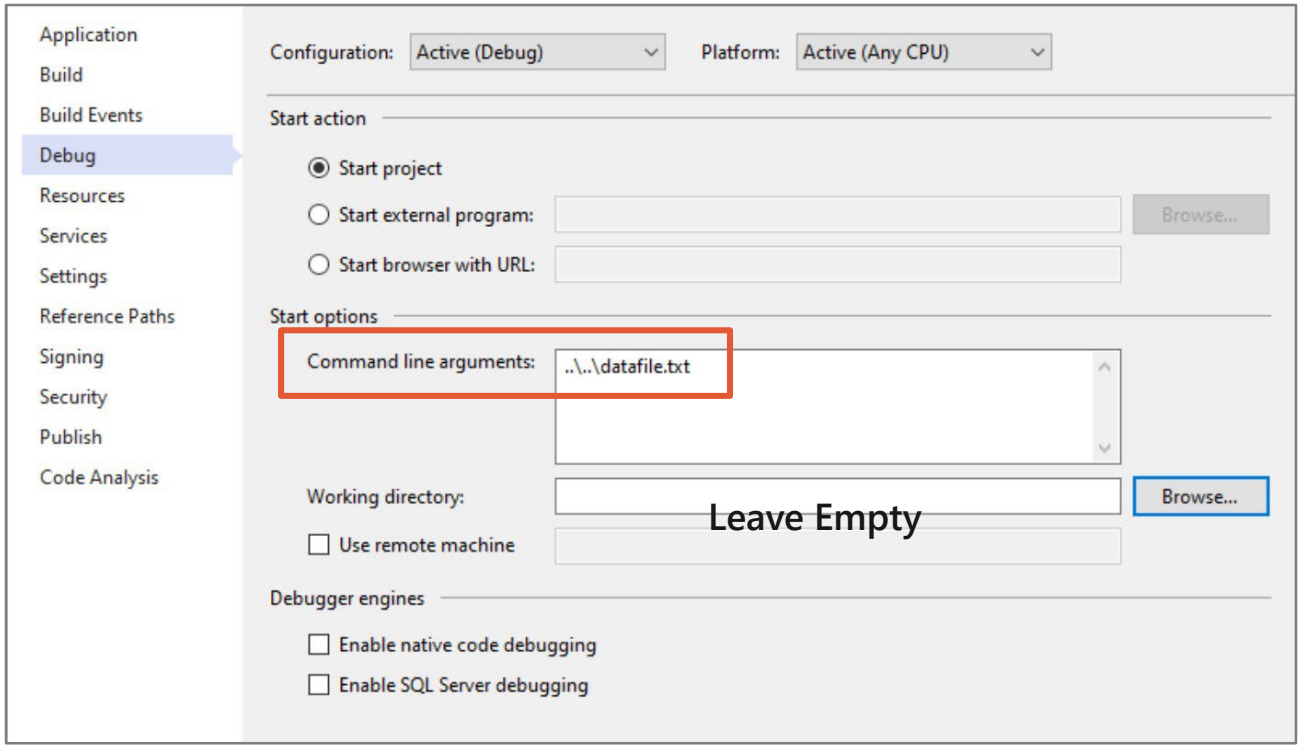
```
D:\Work>helloworld Hello World "a b"
args[0] == Hello
args[1] == World
args[2] == a b

D:\Work>
```

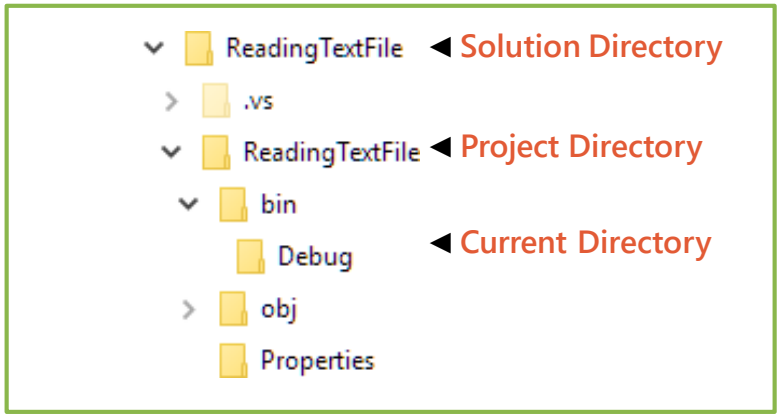
Four possible signatures:

```
static void Main(string[] args)
static void Main()
static int Main(string[] args)
static int Main()
```

Setting Command Line Arguments in Visual Studio

Click Project ▶ Properties





Your program will look for the data file in the Current Directory. The ideal location for your data file is in the Project Directory which is two levels up. Use a relative path to access:

```
..\..\datafile.txt
```