Equals vs ==

Equals vs ==

- Most languages have two ways to compare variables for equality
- Value types:
  - == **&** Equals return true if values are equal (but using == is more common)
- Reference types:
  - == compares object references → are these the same objects?
  - Equals compares values → do these objects have the same values?
- Exceptions:
  - string is a reference type with value type semantics
    in other words == is implement based on string value comparison
  - Object.Equals behaves as == by default

Equals vs ==

- Equals is a method that can be overridden by subclasses:
  - Which Equals implementation is chosen is based on polymorphism, specific object type at ***runtime***
- == is an operator that can be overloaded by classes:
  - Which == operator definition is chosen is based on ***compile*** time type

- int, bool, float, etc → ==
- objects → references → ==, values → Equals
- strings → == is usually safe, Equals also works ☺

What does this print?

```
class Program
{
    static void Main(string[] args)
    {
        string stringA = "hello";
        string stringB = "hello";
        string stringC = new string(new char[] { 'h', 'e', 'l', 'l', 'o'});

        System.Console.WriteLine(stringA == stringB);
        System.Console.WriteLine(stringA == stringC);

        System.Console.ReadLine();
    }
}
```

Output:
True
True

What does this print?

```
class Program
{
    static void Main(string[] args)
    {
        object stringA = "hello";
        object stringB = "hello";
        object stringC = new string(new char[] { 'h', 'e', 'l', 'l', 'o'});

        System.Console.WriteLine(stringA == stringB);
        System.Console.WriteLine(stringA == stringC);

        System.Console.ReadLine();
    }
}
```

Output:
True
False

What does this print?

```
class Program
{
    static void Main(string[] args)
    {
        System.Console.WriteLine("hello" == "hello");
        System.Console.WriteLine((object)"hello" == "hello");
        System.Console.WriteLine("hello" == new string(new char[] { 'h', 'e', 'l', 'l', 'o' }));
        System.Console.WriteLine((object)"hello" == new string(new char[] { 'h', 'e', 'l', 'l', 'o' }));

        System.Console.ReadLine();
    }
}
```

Output:
True
True
True
False

What does this print?

```
class Program
{
    static void Main(string[] args)
    {
        object stringA = "hello";
        object stringB = "hello";
        object stringC = new string(new char[] { 'h', 'e', 'l', 'l', 'o'});

        System.Console.WriteLine(stringA.Equals(stringB));
        System.Console.WriteLine(stringA.Equals(stringC));

        System.Console.ReadLine();
    }
}
```

Output:
True
True

What does this print?

```
class Program
{
    class Test { }

    static void Main(string[] args)
    {
        object objectA = new Test();
        object objectB = new Test();

        System.Console.WriteLine(objectA == objectB);
        System.Console.WriteLine(objectA.Equals(objectB));

        System.Console.ReadLine();
    }
}
```

Output:
False
False

What does this print?

```
class Program
{
    struct Test { }

    static void Main(string[] args)
    {
        object objectA = new Test();
        object objectB = new Test();

        System.Console.WriteLine(objectA == objectB);
        System.Console.WriteLine(objectA.Equals(objectB));

        System.Console.ReadLine();
    }
}
```

Output:
False
True

What does this print?

```
class Program
{
    struct Test { }

    static void Main(string[] args)
    {
        Test objectA = new Test();
        Test objectB = new Test();

        System.Console.WriteLine(objectA == objectB);
        System.Console.WriteLine(c

        System.Console.ReadLine();
    }
}
```

(local variable) Test objectA

Operator '==' cannot be applied to operands of type 'Program.Test' and 'Program.Test'

Equals vs ==

- Moral of the story, in theory:
  - == → references (shallow comparison)
  - Equals →
    - By default by value (deep comparison) for structs
    - By default by reference (shallow comparison) for classes
  - == can be overloaded, Equals can be overridden
- There are some weird exceptions to the general rules based on compile type declarations / struct vs classes

- Main thing: be aware of these two comparison methods and that you might be using the **wrong** one

List.IndexOf / Contains revisited

List.IndexOf and List.Contains

- Lecture 2 recap, pseudocode:

List.Contains (item) pseudo:

```
return List.IndexOf (item) > -1
```

List.IndexOf (item) pseudocode*:

```
for (int i = 0; i < Count;++i) {
        if internalArray[i] equals item return i;
}
return -1;
```

- The equals in the pseudo code is actually implemented using Equals, not ==
  (This also means you can override Equals and wreak havoc on your list ☺)

All the dirty details

- https://stackoverflow.com/questions/814878/c-sharp-difference-between-and-equals
- https://blogs.msdn.microsoft.com/ericlippert/2009/04/09/double-your-dispatch-double-your-fun/
- https://devblogs.microsoft.com/csharpfaq/when-should-i-use-and-when-should-i-use-equals/
- https://coding.abel.nu/2014/09/net-and-equals/
- https://docs.microsoft.com/en-us/previous-versions/ms173147(v=vs.90)
- https://medium.com/@equisept/c-journey-into-struct-equality-comparison-deep-dive-9693f74562f1