Learn C#: References

C# Reference Types

In C#, classes and interfaces are *reference types*. Variables of reference types store references to their data (objects) in memory, and they do not contain the data itself.

An object of type `Object` , `string` , or `dynamic` is also a reference type.

```
SportsCar sc = new SportsCar(100);
SportsCar sc2 = sc;
sc.SpeedUp(); // Method adds 20
Console.WriteLine(sc.Speed); // 120
Console.WriteLine(sc2.Speed); // 120

// In this code, sc and sc2 refer to the same
object. The last two lines will print the same
value to the console.
```

C# Object Reference

In C#, an object may be referenced by any type in its inheritance hierarchy or by any of the interfaces it implements.

```
// Woman inherits from Human, which inherits
from Animal, and it implements IPerson:
class Human : Animal
class Woman : Human, IPerson

// All of these references are valid:
Woman eve = new Woman();
Human h = eve;
Animal a = eve;
IPerson p = eve;
```

C# Object Reference Functionality

In C#, the functionality available to an object reference is determined by the reference's type, not the object's type.

```
Player p = new Player();
Fan f = p;
p.SignContract();
f.SignContract();
// Error! 'SignContract()` is not defined for
the type 'Fan'
```

C# Polyphormism

Polymorphism is the ability in programming to present the same interface for different underlying forms (data types). We can break the idea into two related concepts. A programming language supports *polymorphism* if:

1. Objects of different types have a common interface (interface in the general meaning, not just a C# interface), and
2. The objects can maintain functionality unique to their data type

```
class Novel : Book
{
  public override string Stringify()
  {
    return "This is a Novel!;
  }
}

class Book
{
  public virtual string Stringify()
  {
    return "This is a Book!;
  }
}

// In the below code, you’ll see that a Novel
and Book object can both be referred to as
Books. This is one of their shared interfaces.
At the same time, they are different data
types with unique functionality.

Book bk = new Book();
Book warAndPeace = new Novel();
Console.WriteLine(bk.Stringify());
Console.WriteLine(warAndPeace.Stringify());

// This is a Book!
// This is a Novel

// Even though bk and warAndPeace are the same
type of reference, their behavior is
different. Novel overrides the Stringify()
method, so all Novel objects (regardless of
reference type) will use that method.
```

C# Upcasting

In C#, *upcasting* is creating an inherited superclass or implemented interface reference from a subclass reference.

```
// In this case, string inherits from Object:

string s = "Hi";
Object o = s;

// In this case, Laptop implements the
IPortable interface:

Laptop lap = new Laptop();
IPortable portable = lap;
```

C# Downcasting

In C#, *downcasting* is creating a subclass reference from a superclass or interface reference.
Downcasting can lead to runtime errors if the superclass cannot be cast to the specified subclass.

```
Account a = new Account();
CustomerAccount ca = a;
// error CS0266: Cannot implicitly
convert type `Account` to
`CustomerAccount`. An explicit
conversion exists (are you missing
a cast?)
```

```
// Dog inherits from Pet. An implicit downcast
throws a compile-time error:
Pet pet = new Pet();
Dog dog = pet;
// error CS0266: Cannot implicitly convert
type `Pet` to `Dog`. An explicit conversion
exists (are you missing a cast?)

// Every downcast must be explicit, using the
cast operator, like (TYPE). This fixes the
compile-time error but raises a new runtime
error.
Pet pet = new Pet();
Dog dog = (Pet)pet;
// runtime error: System.InvalidCastException:
Specified cast is not valid.

//The explicit downcast would only work if the
underlying object is of type Dog:
Dog dog = new Dog();
Pet pet = dog;
Dog puppy = (Dog)pet;
```

C# Null Reference

In C#, an undefined reference is either a *null reference* or *unassigned*. A null reference is represented by the keyword `null`.
Be careful when checking for null and unassigned references. We can only compare a null reference if it is explicitly labeled `null`.

```
MyClass mc; //unassigned

Console.WriteLine (mc == null);
// error CS0165: Use of unassigned local
variable 'mc'

MyClass mc = null; //explicitly 'null'

Console.WriteLine(mc == null);
// Prints true.

// Array of unassigned references
MyClass[] objects = new MyClass[5];
// objects[0] is unassigned, objects[1] is
unassigned, etc...
```

C# Value Types

In C#, value types contain the data itself. They include `int`, `bool`, `char`, and `double`.
Here's the entire list of value types:

- `char`, `bool`, `DateTime`
- All numeric data types
- Structures (`struct`)
- Enumerations (`enum`)

C# Comparison Type

In C#, the type of comparison performed with the equality operator (`==`), differs with reference and value types.

When two value types are compared, they are compared for *value equality*. They are equal if they hold the same value.

When two reference types are compared, they are compared for *referential equality*. They are equal if they refer to the same location in memory.

```
// int is a value type, so == uses value
equality:
int num1 = 9;
int num2 = 9;
Console.WriteLine(num1 == num2);
// Prints true

// All classes are reference types, so == uses
reference equality:
WorldCupTeam japan = new WorldCupTeam(2018);
WorldCupTeam brazil = new WorldCupTeam(2018);
Console.WriteLine(japan == brazil);
// Prints false
// This is because japan and brazil refer to
two different locations in memory (even though
they contain objects with the same values):
```

C# Override

In C#, the `override` modifier allows base class references to a derived object to access derived methods.
In other words: If a derived class overrides a member of its base class, then the overridden version can be accessed by derived references AND base references.

```
// In the below example,
DerivedClass.Method1() overrides
BaseClass.Method1(). bcdc is a BaseClass-type
reference to a DerivedClass value. Calling
bcdc.Method1() invokes DerivedClass.Method1().

class MainClass {
  public static void Main (string[] args) {
    BaseClass bc = new BaseClass();
    DerivedClass dc = new DerivedClass();
    BaseClass bcdc = new DerivedClass();

    bc.Method1();
    dc.Method1();
    bcdc.Method1();
  }
}

class BaseClass
{
    public virtual void Method1()
    {
        Console.WriteLine("Base - Method1");
    }
}

class DerivedClass : BaseClass
{
    public override void Method1()
    {
        Console.WriteLine("Derived
- Method1");
    }
}

// The above code produces this result:
// Base - Method1
// Derived - Method1
// Derived - Method1

// If we wanted bcdc.Method1() to invoked
BaseClass.Method1(), then we would label
DerivedClass.Method1() as new, not override.
```

C# Object Class

In C#, the base class of all types is the `Object` class. Every class implicitly inherits this class.
When you create a class with no inheritance, C# implicitly makes it inherit from `Object`.

```
// When you write this code:
class Dog {}
// C# assumes you mean:
class Dog : Object {}

//Even if your class explicitly inherits from
a class that is NOT an Object, then some class
in its class hierachy will inherit from
Object. In the below example, Dog inherits
from Pet, which inherits from Animal, which
inherits from Object:
class Dog : Pet {}
class Pet : Animal {}
class Animal {}

//Since every class inherits from Object, any
instance of a class can be referred to as an
Object.
Dog puppy = new Dog();
Object o = puppy;
```

C# Object Class Methods

In C#, the `Object` class includes definitions for these methods: `ToString()` , `Equals(Object)` , and `GetType()` .

```
Object obj = new Object();
Console.WriteLine(obj.ToString());
// The example displays the following output:
//      System.Object

public static void Main()
{
    MyBaseClass myBase = new MyBaseClass();
  MyDerivedClass myDerived = new
MyDerivedClass();
  object o = myDerived;
  MyBaseClass b = myDerived;

    Console.WriteLine("mybase: Type is {0}",
myBase.GetType());
  Console.WriteLine("myDerived: Type is {0}",
myDerived.GetType());
  Console.WriteLine("object o = myDerived:
Type is {0}", o.GetType());
  Console.WriteLine("MyBaseClass b =
myDerived: Type is {0}", b.GetType());
}

// The example displays the following output:
//    mybase: Type is MyBaseClass
//    myDerived: Type is MyDerivedClass
//    object o = myDerived: Type is
MyDerivedClass
//    MyBaseClass b = myDerived: Type is
MyDerivedClass
```

C# ToString() Method

When a non-string object is printed to the console with `Console.WriteLine()` , its `ToString()` method is called.

```
Random r = new Random();

// These two lines are equivalent:
Console.WriteLine(r);
Console.WriteLine(r.ToString());
```

C# String Comparison

In C#, `string` is a reference type but it can be compared by value using `==` .

```
//In this example, even if s and t are not
referentially equal, they are equal by value:
string s = "hello";
string t = "hello";

// b is true
bool b = (s == t);
```

C# String Types Immutable

In C#, `string` types are *immutable*, which means they cannot be changed after they are created.

```
// Two examples demonstrating how
immutablility determines string behavior. In
both examples, changing one string variable
will not affect other variables that
originally shared that value.

//EXAMPLE 1
string a = "Hello?";
string b = a;
b = "HELLLLLLLO!!!!";

Console.WriteLine(b);
// Prints "HELLLLLLLO!!!!"

Console.WriteLine(a);
// Prints "Hello?"


//EXAMPLE 2
string s1 = "Hello ";
string s2 = s1;
s1 += "World";

System.Console.WriteLine(s2);
// Prints "Hello "
```

C# Empty String

In C#, a `string` reference can refer to an empty string with `""` and `String.Empty`.
This is separate from `null` and unassigned references, which are also possible for `string` types.

```
// Empty string:
string s1 = "";

// Also empty string:
string s2 = String.Empty;

// This prints true:
Console.WriteLine(s1 == s2);

// Unassigned:
string s3;

// Null:
string s4 = null;
```