Value and Reference Types

C# Programming

In this chapter we will learn ...

- About value types and how to create them
- About reference types and how to create them

Value Types

C# Value Types include

- int, byte, short, unit64, uint16 etc
- float, double, decimal
- bool
- char

Usually small data types

- Used as fields in other types
- Local variables of functions

Value Types - Quick Facts

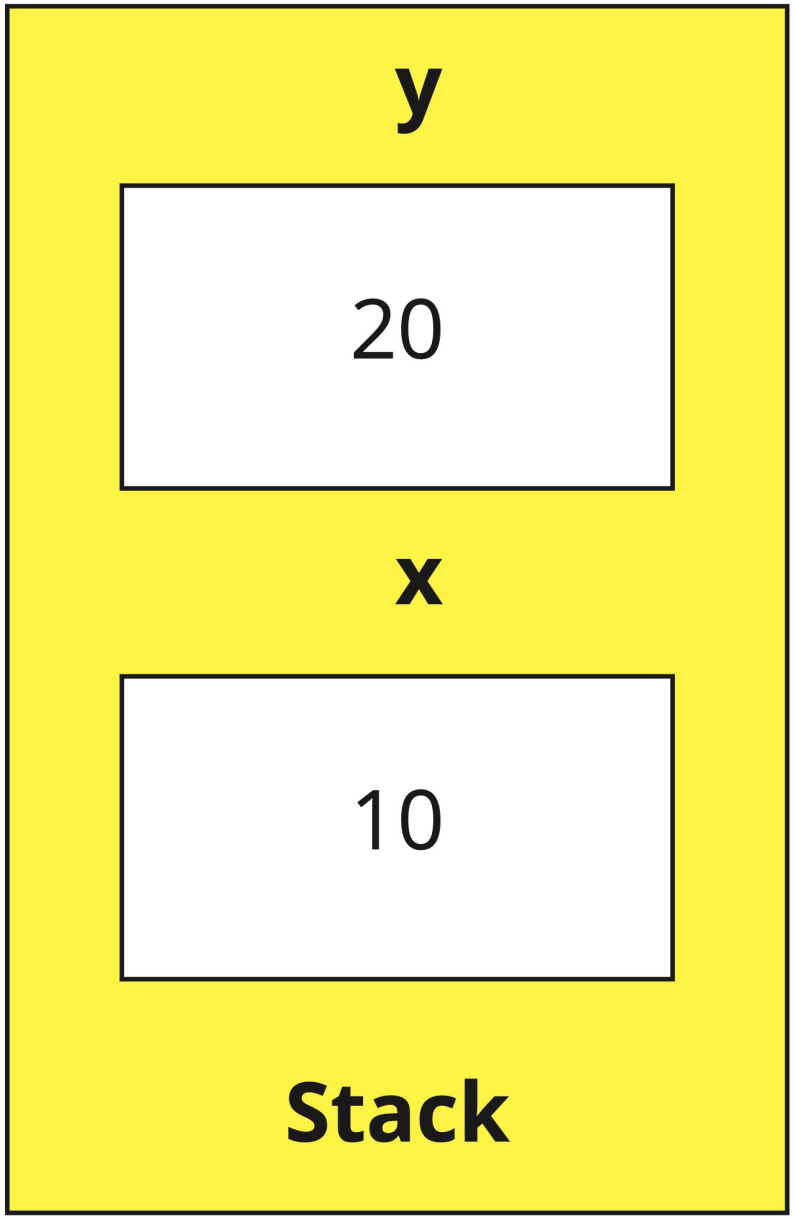
- Allocated on the stack when declared with a function
- Defined by structs, tuples or enums
- Assigning one value type variable to another always creates a copy
- Do not need Garbage collection
    - (Can be attributes of an object which do)
- Don't support inheritance

Value Types Memory Allocation

- Added to the stack as local variables in functions
- Allocated to the heap as member variables of classes

```
void DoSomething() {
    int x = 10;
    int y = 20;
}
```

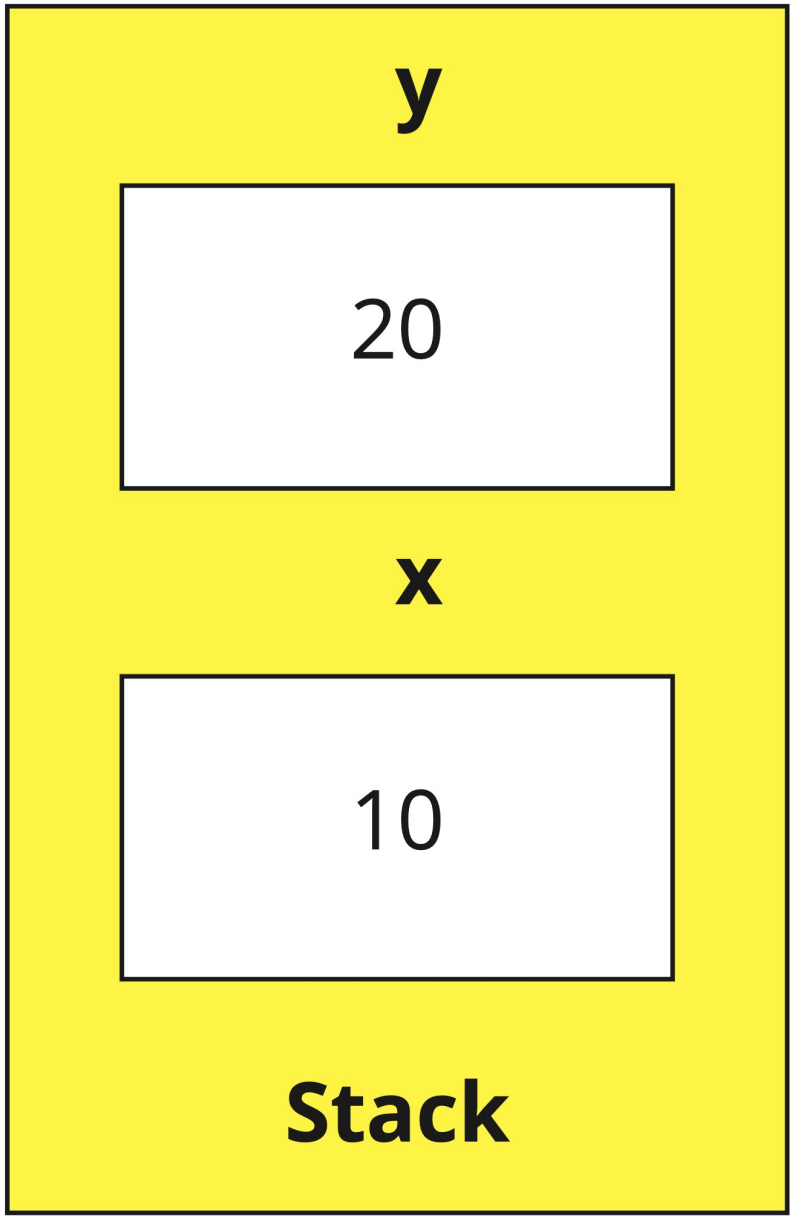
- Value is held within variable



Value Type Stack Variables

- Pushed onto the stack when a function executes
- Pops off the stack once function has finished executing
- Simple memory model

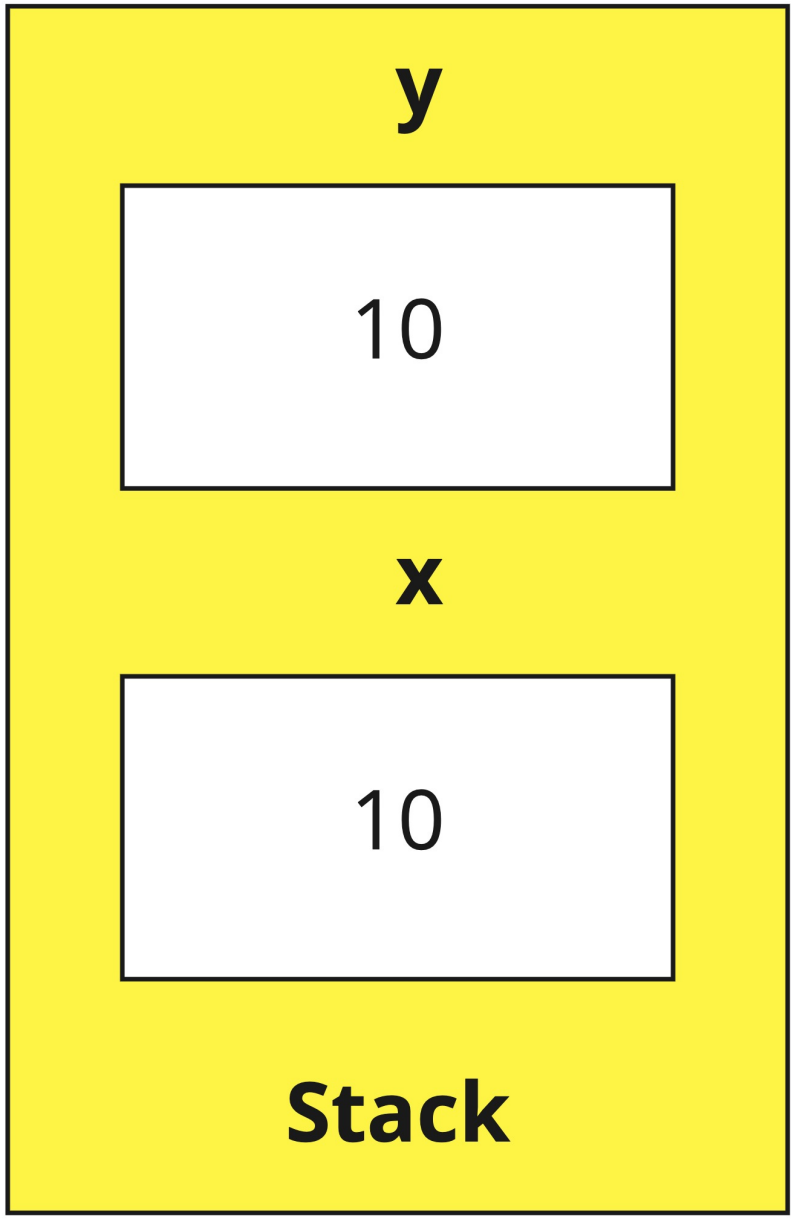
```
void DoSomething() {
    int x = 10;
    int y = 20;
}
```



Assigning a value type creates an independent copy

```
void DoSomething() {
    int x = 10;
    int y = x;
}
```

- x and y are independent of each other
- Changing x will not affect y

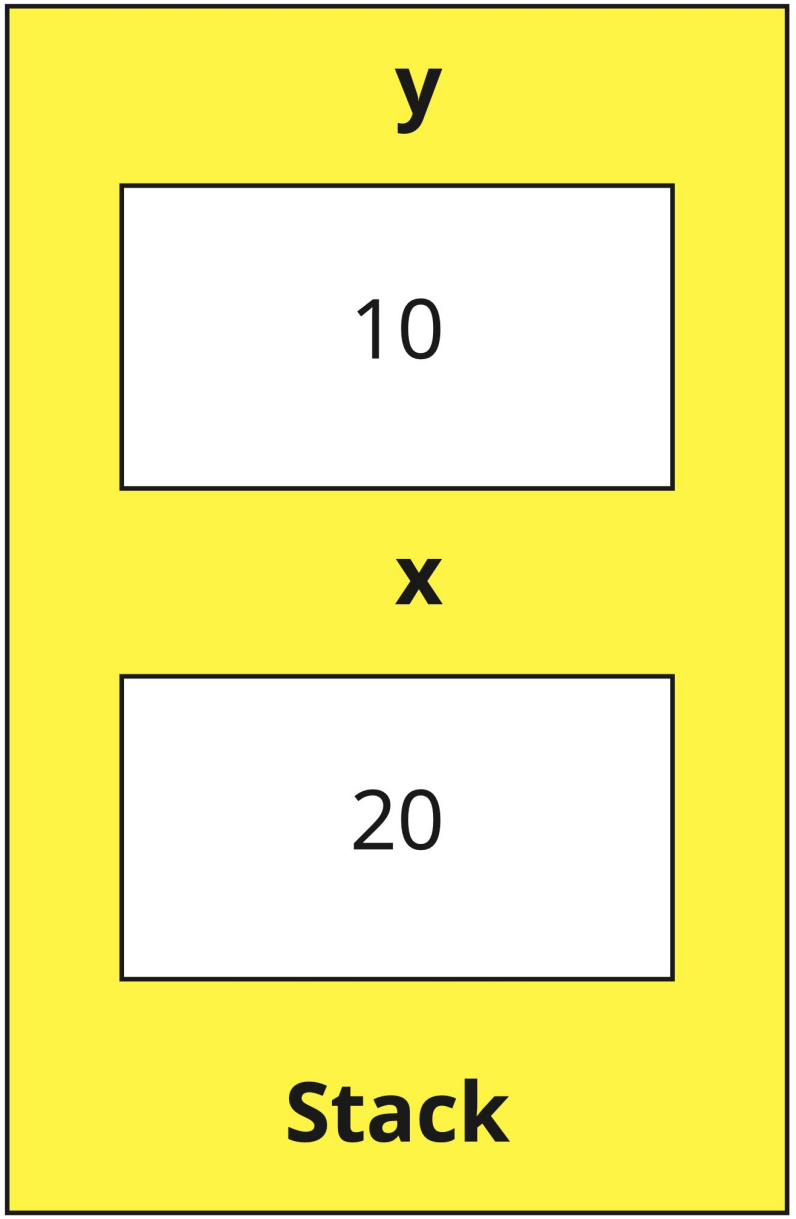


Independent copies don't affect each other

```
void DoSomething() {
    int x = 10;
    int y = x;
    x = 20;
}
```

- x changes to 20
- y remains at 10. No side effect!

Value Types have no side effects



structs

- Are Value Types
- Support
  - Fields
  - Methods
  - Properties
  - Events
- Don't support inheritance
- Can implement one or more interfaces
  - Support Composition
- Used for small attribute style data types

Defining new Value Types with struct

```
public struct Rectangle
{
    public int Width ;
    public int Height ;
    public int Area => Width * Height;
}
```

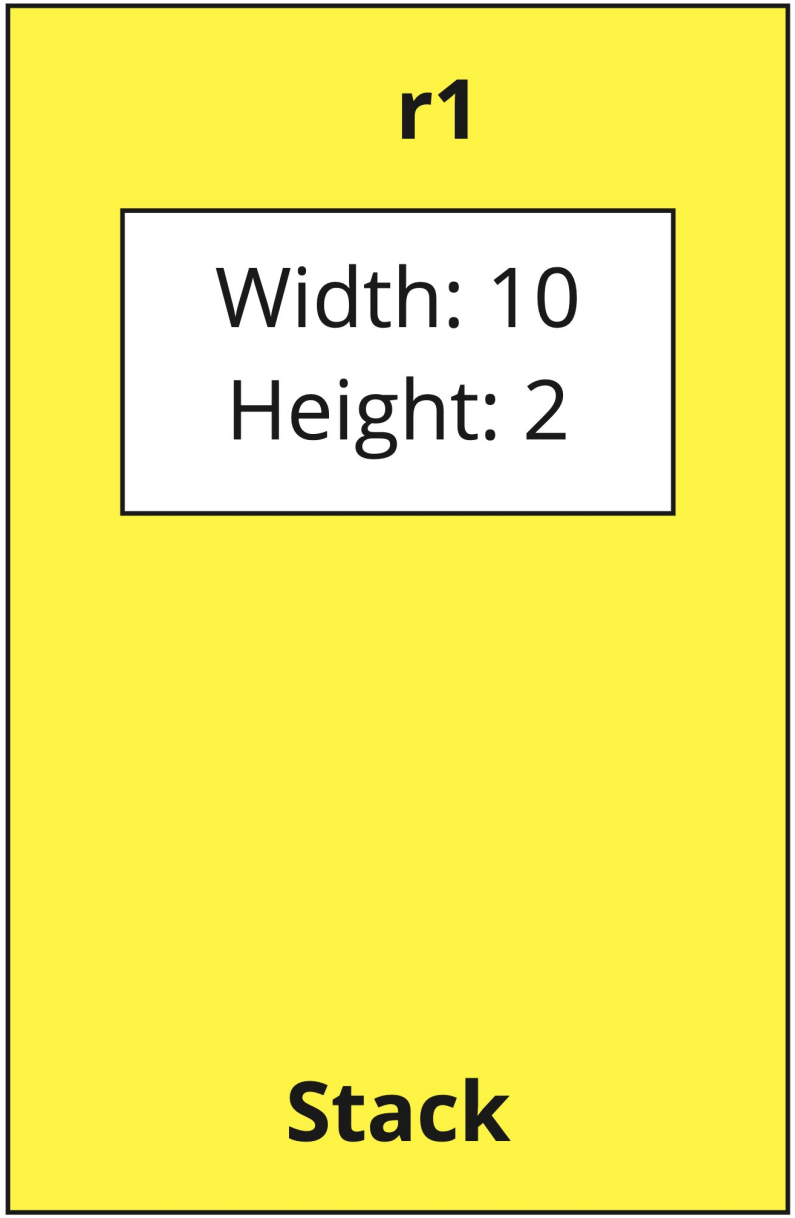
- Defines a value type containing two fields
  - Width as an int
  - Height as an int

Declaring a custom Value Type

```
void DoSomething() {
    Rectangle r1; // Declare variable
    r1.Width = 10; // Set field values
    r1.Height = 2;
    Console.WriteLine(r1.Area);
}
```

When DoSomething() executes

- Rectangle will be allocated onto stack
- Will contain Width and Height fields
- Will print 20



Initializing structs

structs must be initialized before access

- Directly or via Properties, Methods, Events

```
void DoSomething() {
    Rectangle r1; // Declare variable
    r1.Width = 10; // Set field values
    // r1.Height = 2;
    Console.WriteLine(r1.Area); // Error: Unassigned Variable
}
```

- All fields must be initialized first

Using Constructor syntax

structs can have constructor functions

- Allows struct to be initialized with starter values

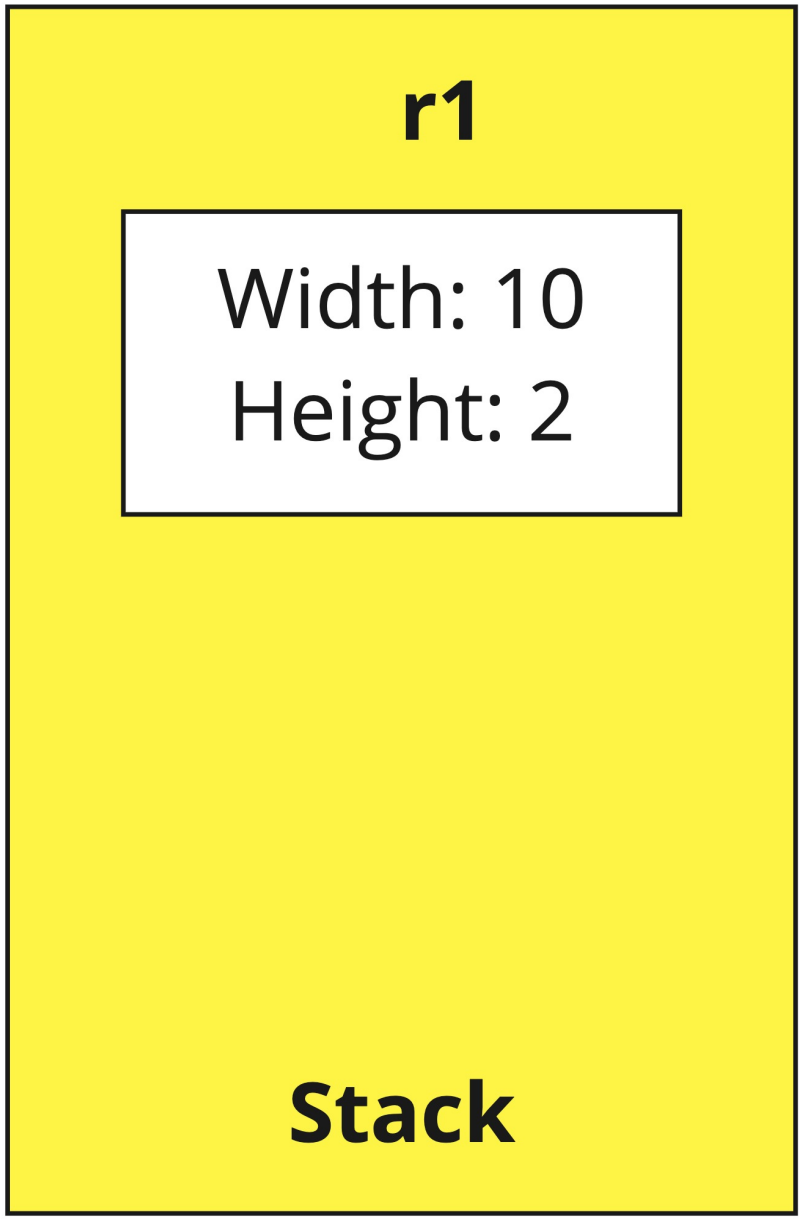
```
public struct Rectangle {
    public int Width ;
    public int Height ;
    public int Area => Width * Height;
    public Rectangle(int width, int height) {
        Width = width;
        Height = height;
    }

}
```

Using the struct constructor

```
Rectangle r1 = new Rectangle(10, 2);
Console.WriteLine(r1.Area);
```

- Value Type behaves as normal
  - Allocated onto stack if declared in a function
- Use of new invokes constructor
- No objects are created on Heap



Constructor must initialize All fields of struct

```
struct Rectangle
{
        public int Width;
        public int Height;
        public int X, Y;
        public int Area => Width * Height;
        public Rectangle(int width, int height) // Error - Must Initialise X and Y
        {
            Width = width;
            Height = height;
        }

}
```

- struct fields can be initialized at declaration
    - C# 10 onwards

Blank (parameterless) Constructor - C# 10

Structs can have a parameterless/blank constructor

- You must initialize all fields

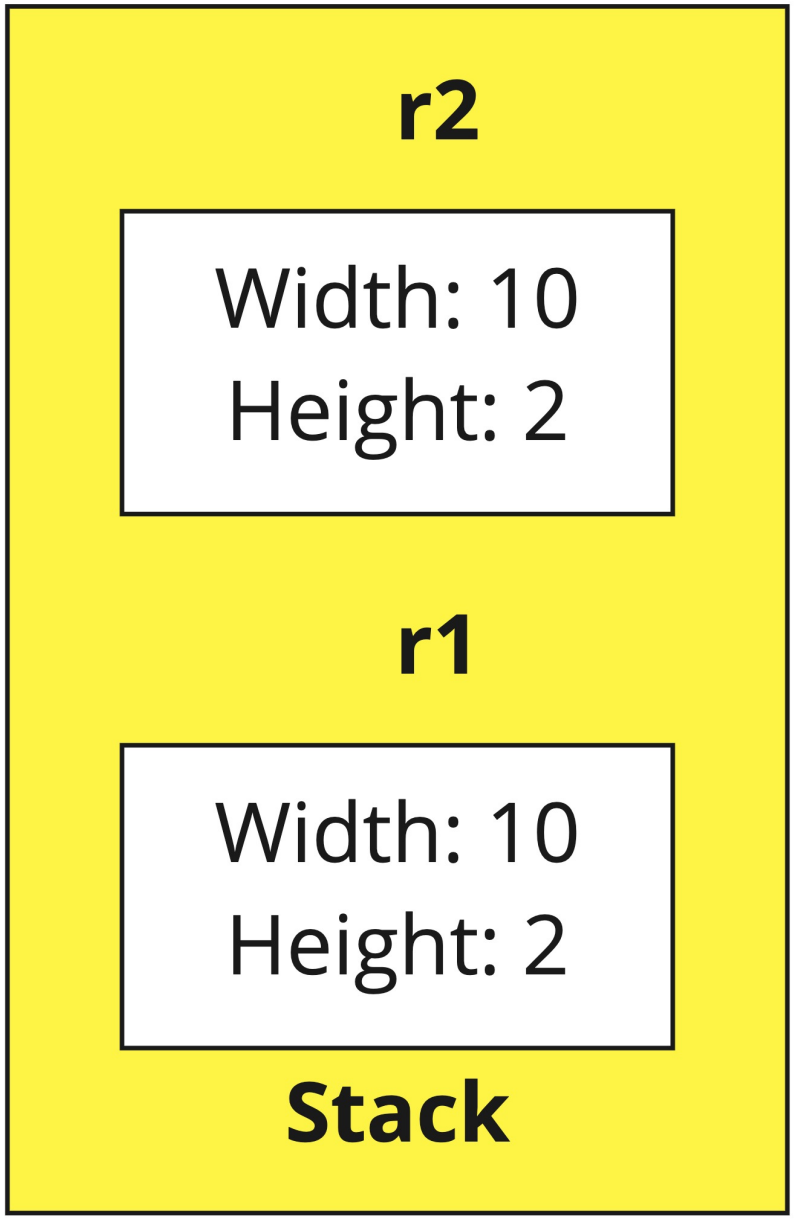
```
...
public Rectangle()
{
    Width = 0;
    Height = 0;
}
```

Copying structs

structs will be copied on assignment

```
> Rectangle r1 = new (10, 2);
> Rectangle r2 = r1;
> r1.Area
20
> r2.Area
20
```

- r2 is distinct copy of r1

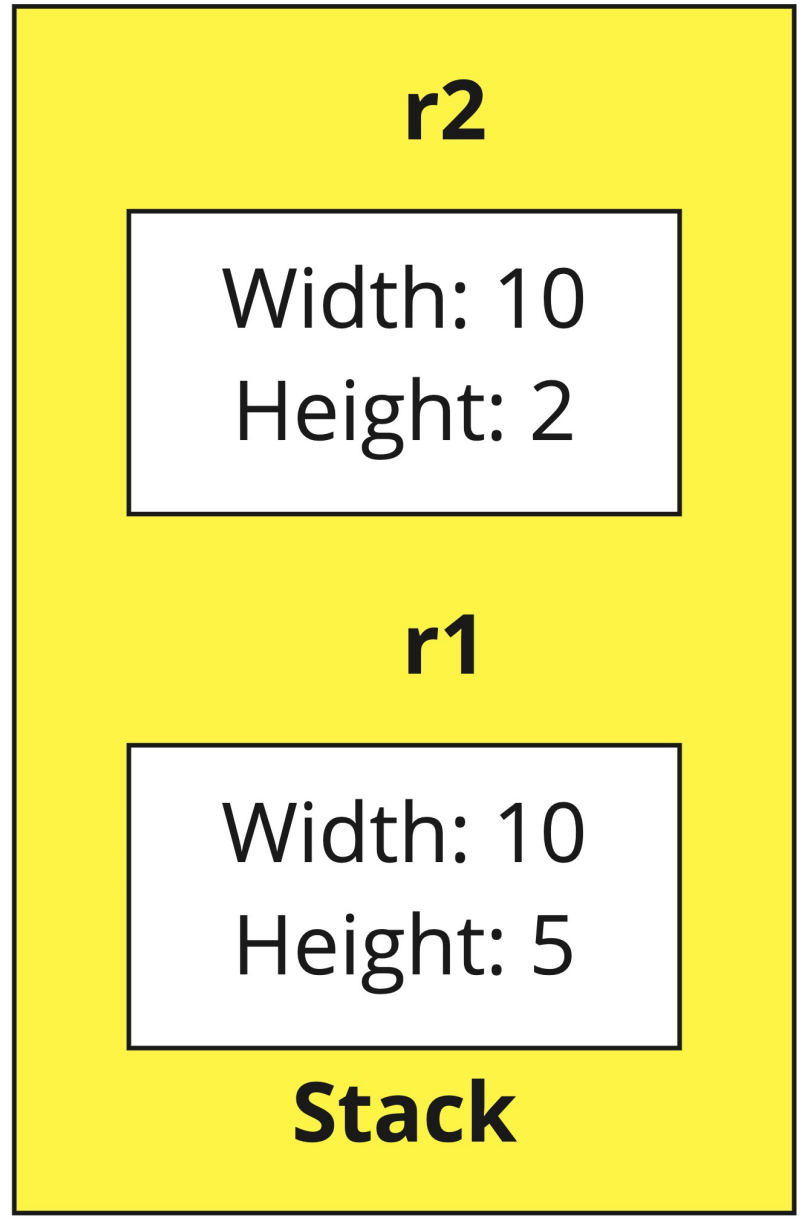


Recap

Value Types have limited side effects

```
> Rectangle r1 = new (10, 2);
> Rectangle r2 = r1;
> r1.Height = 5;
> r1.Area
50
> r2.Area
20
```

- Changing r1 does not impact r2



Checking struct equality

structs can be tested for equality using Equals method

```
> Rectangle r1 = new (10, 2);
> Rectangle r2 = r1;
> r1.Equals(r2)
true
> r1.Height = 5;
> r1.Equals(r2)
false
```

- Cannot use == or != unless operator overloading is implemented

Mutable structs

Mutable types allow their attributes to change

```
struct Rectangle {
        public int Width {get; set;} // Property with setter
        public int Height {get; set;} // Property with setter
        public Rectangle(int width, int height) {
            Width = width;
            Height = height;
        }
        public void Scale(int by) {
            Width *= by;
            Height *= by;
        }
}
```

Immutable structs

structs can be declared as immutable

- Type is declared as readonly

When you declare a type as readonly

- All fields must be declared readonly
- All properties must be declared readonly

Example - Immutable structs

```
readonly struct Rectangle {
        public readonly int Width {get; init;} // Property with init only setter
        public readonly int Height {get; init;} // Property with init only setter
        public Rectangle(int width, int height) {
            Width = width;
            Height = height;
        }
        public Rectangle Scale(int by) {
            // A new Rectangle is returned
            return new Rectangle(Width * by, Height * by);
        }
        public int Area => Width * Height;
}
```

- Notice the init only properties
- Scale method returns a new Rectangle. Does not mutate existing values

Non Destructive mutation

By

- Declaring a struct with a parameterless constructor
- Using a with operator

Non destructive mutation can be achieved

```
> Rectangle r1 = new Rectangle(10, 2);
> Rectangle r2 = with r1 { Height = 5}; // Copy r1, mutate Height to 5
r2.Area
```

Flexible structs

- Consider the following struct

```
struct AccountNumber {
        int number;
        public AccountNumber(int number){
            this.number = number;
        }
}
```

```
AccountNumber acc = new AccountNumber(1234); // Works!
AccountNumber num = 1234; // Fails!!!!
```

Implicit conversion

- You must add implicit conversion operators

```
struct AccountNumber {
        int number;
        public AccountNumber(int number){
            this.number = number;
        }
        public static implicit operator AccountNumber(int value) {
            if (value.ToString().Length == 7)
                return new AccountNumber(value);
            else
                throw new ArgumentException("Not a valid Account number");
        }
    }
```

Implicit Casting

- How do we retrieve an int from an AccountNumber?

```
AccountNumber acc = new AccountNumber(1234);
int num = acc;
```

- Implement an implicit cast operator for **int**

```
struct AccountNumber {
    ...
    public static implicit operator int(AccountNumber acc) => acc.number;
}
```

ref struct Types

Force allocation of structs onto the stack only

```
ref struct Rectangle {

}
```

That means ref structs ...

- Can't be fields/properties of classes
- Cant be declared as an array or with a generic type
- Can't implement interfaces
- Can't be function arguments

struct recommendations

1. Keep them small - Ideally <= 16-32 bytes
    - They have to be allocated onto the stack
    - They always get copied
2. Use them for genuine added value attribute types
    - AccountNumbers, ZipCodes etc
3. Use with Interfaces not Inheritance

Reference Variables

References essentially

- Reference objects
- Are accessible from managed code
- Auto dereference - no need for dereferencing operator

```
int x = 10;
ref int p = ref x; // Get reference to x
Console.WriteLine(p); // Can print p which will auto reference to x to print 10
```

Reference Types - Quick Facts

- Defined by two things
  - A reference variable that points to
  - On Object that is allocated on the heap
- Defined by classes
- Assigning one reference type variable to another only copies the reference NOT the object
- Need Garbage collection
- Support inheritance

Classes and Objects

Reference Types are defined by classes (or records)

Class

- Defines a template for creating objects on the heap
    - Defines attributes, methods, properties and events

Object

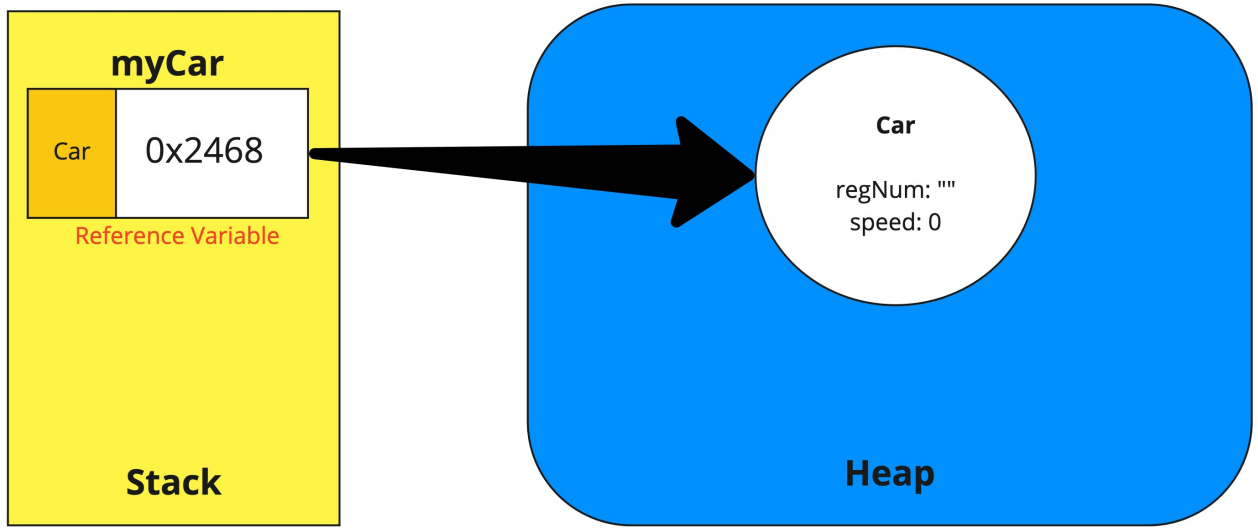
- An instance of a class in memory
- Occupies its own memory location
- Has own unique identity and separate set of attribute values

Example - Classes

```
class Car {
    public int speed;
    public string regNum;
}
```

- Object created from class

```
// Reference        Object
Car myCar      =   new Car();
```



Anatomy of a Reference Type

Reference Types have

- A reference variable
  - Holds a reference/address of the object on the heap
- An object that stores the data defined by the attributes class
  - Object is stored at a specific memory address on the heap

Reference variables act as small lightweight pointers

- References can be passed around app regardless of size of object

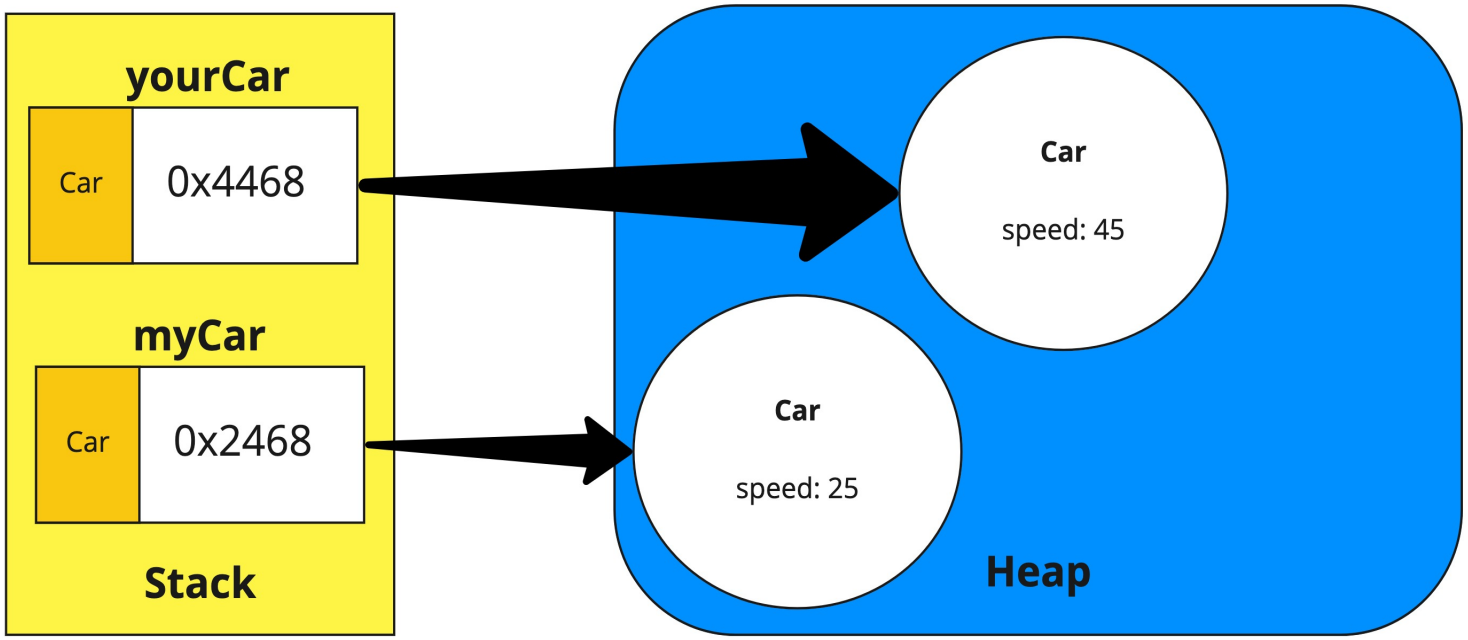
Accessing objects

Access to the object requires the reference variable

```
Car myCar = new Car();
myCar.speed = 25;
```

Working with References

```
Car myCar = new Car();
myCar.speed = 25;
Car yourCar = myCar;
myCar.speed = 45;
```



Assigning References

Assigning one reference to another

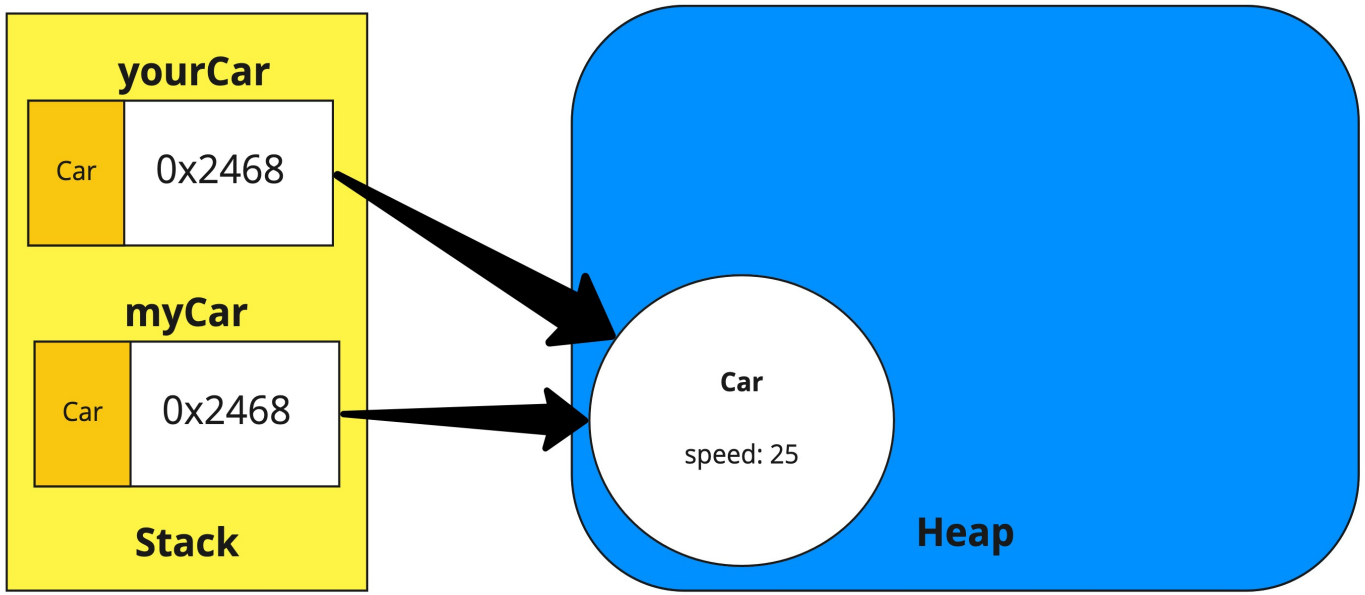
- Copies the reference not the object
- Result is two reference both referencing the same object

Reference Assignment creates side effects

- What you do to one of the reference copies applies to the common object
- Effect will be felt by other reference

Assigning References - Example

```
> Car myCar = new Car();
> Car yourCar = myCar;
> myCar.speed = 25;
> yourCar.speed
25
```



Initializing Classes

Data Fields can be initialized

- At declaration
- Inside one or more constructors
- Auto Initialized to the types default value (null or zero)

Auto Initialization

Fields will be initialized to the default values for their type

- 0 for numbers
- false for bool
- null for reference types like strings

```
public class Car {
    public int speed ; // Initialised to 0
    public float
}
```

Declarative Initialization

Fields can be explicitly initialized at declaration

```
public class Car {
    public int speed = 0; // Initialised explictly to 0
}
```

Initialization via Constructor

Constructors functions explicitly initialize an object

- Don't need to initialize all fields

```
public class Car {
    public int speed ;
    // Empty constructor function has no parameters
    public Car(){
        this.speed = 0;
    }
}
```

- An empty constructor will be generated if no other constructor is defined

Overloaded Constructors

Multiple constructor functions can be defined

```
public class Car {
    public int speed ;
    public Car(){
        this.speed = 0;
    }
    public Car(int speed){
        this.speed = speed;
    }
}
```

Invoking Constructors

- Constructor invoked on object creation

```
> Car myCar = new Car(); // Invokes the blank constructor
> Car yourCar = new Car(5); // Invokes the Car(int speed) constructor
> myCar.speed
0
> yourCar.speed
5
```

Chaining Constructor Calls

One constructor can be called from another

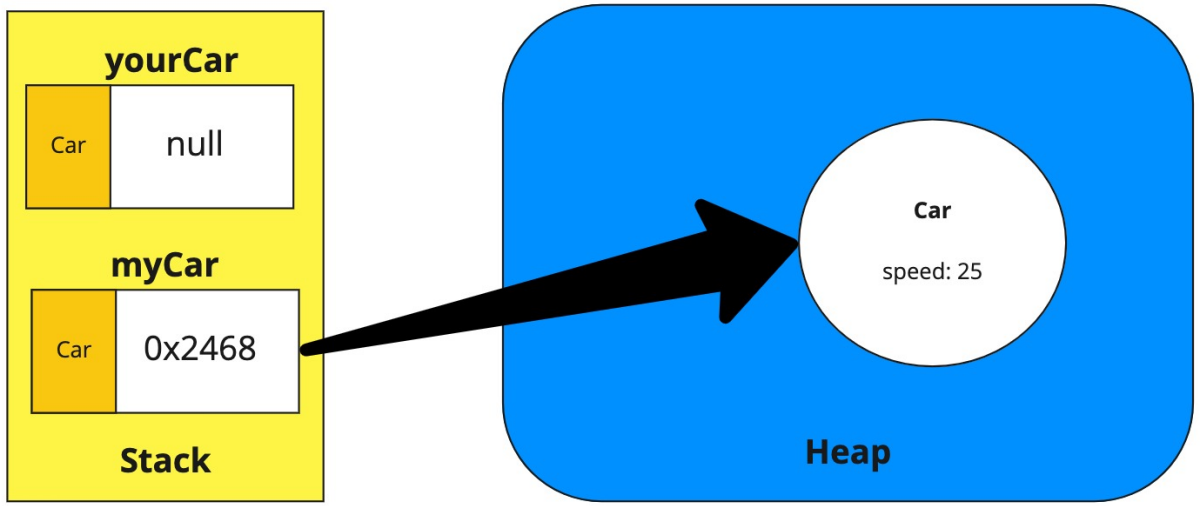
- Can be used to reuse initialization code
- Use **this** keyword to call to other constructor

```
public class Car {
    public int speed ;
    public Car(): this(0) {
    }
    public Car(int speed){
        this.speed = speed;
    }
}
```

null and reference types

So what is null

- When reference isn't pointing to an object
  - null is a default literal value stored in reference variables
- null generally means empty or unknown



nullable value types - C# 7.0

How can we represent unknown or empty with a value type variable?

- Declare a value type as nullable using
  - ?
  - Generic Nullable

```
> int? age = null;
> Nullable<decimal> price = 5.75;
> age
null
> age == null && price != null
true
```

- A base type can be assigned to it's nullable counter-part

Working with nullable

Nullable types provide helper features

- HasValue - true if variable is not null
- Value - Actual non nullable value
    - Nullable returns int

```
Nullable<decimal> price = 5.75;
if (price.HasValue) {
    decimal tax = price.Value * 0.2m;
    Console.WriteLine($"Tax:{tax}");
}
```

Take care with nullable

```
> int? age = null;
> int actualAge = age;
(1,9) error : Cannot implicitly convert from int? to int
```

- Explicit cast is required

```
> int actualAge = (int) age;
```

Safely Working with nullable value types

Using the is operator

- Allows a safe guarded cast

```
int? age = 21;
if (age is int actualAge) {
    Console.WriteLine($"Age is {actualAge}");
}
```

Operators and Nullable Types

Nullable Types will respect standard/overloaded operators

- Operators return null if a Nullable value in evaluation is null

```
> int? a = 10;
> int b = 2;
> a * b
20
> int? c = null;
> b * c
null
```

null operators

null coalescing operator ?? helps deal with null

```
string a = null;
string b = "Hello World";
// If a is not null assign a to c
// Otherwise assign b to c
string c = a ?? b;
```

Dealing with null references

Sometimes you will want to handle a null reference gracefully

```
List<string> names = null;
```

To add a name we could do

```
if (names is null) names = new List<string>();
names.Add("Fred");
```

or use ??= operator

```
(names ??= new List<String>()).Add("Fred"); // if names is null create a new List
```

null coalescing operator

Safely returns null if the expression on the left side of an accessor evaluates to null

Use `?` before `.` accessor

```
Car myCar = null;

int speed = myCar.Speed; // throws Null reference exception

int? speed = myCar?.Speed; // returns null if myCar is null, no exception
```

nullable reference types

Because nulls are responsible for so many Exceptions

We can now outlaw them!

- Change setting in Project file (.csproj)

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    ...
    <Nullable>enable</Nullable>
    ...
  </PropertyGroup>
</Project>
```

- Requires reference types to be initialized and not left null

Equality Testing

Reference Types allow basic equality testing

- Can check if references are equal
    - use == or ReferenceEquals
- Cannot check data field equivalence by default

```
> Car myCar = new Car(5);
> Car yourCar = new Car(5);
> myCar == yourCar
false
```

myCar == yourCar compares references not speeds

Content/Attribute equality

Must be explicitly implemented by overriding methods

- Equals - Explicitly compares attributes of two objects and returns true/false
- GetHashCode - Generates a unique hash code based on attribute combination

Can also override operators

- ==
- != (must be implemented as a pair)

Generating a Hash Code

- Override Equals and GetHashCode
- Use HashCode.Combine to generate a hash-code from attributes

```
class Car {
    int speed;
    string reg;
    public override bool Equals(object obj){
        return this.GetHashCode() == obj.GetHashCode();
    }
    public override int GetHashCode(){
        return HashCode.Combine(speed, reg);
    }
}
```

Overriding comparison operator

You can override == and !=

- Implement Equals and GetHashCode first

```
class Car {
    ...
    public static bool operator == (Car lhs, Car rhs) {
        return lhs.Equals(rhs);
    }
    public static bool operator !=(Car lhs, Car rhs) {
        return !lhs.Equals(rhs);
    }
}
```

- Use **ReferenceEquals()** to test for reference equality after operator override

In this chapter we learned ...

- About value types and how to create them
- About reference types and how to create them