# Inheritance

Object-Oriented Programming

# Topics

- Naming
- Types of Classes
- Deriving
- Constructors
- Upcasting
- Downcasting

# Essential Skill: How to Name Things

- Your code will live longer in the maintenance stage. So, better naming means easier to maintain.
- Why?
  - You likely won’t be the only person touching your code.
  - You likely will forget what you were doing.
- How to name better?
- **Don’t abbreviate**. Which is better? Dept or Department?
- **Make it meaningful**. Which is better? DoIt or MergeSort?
- **Avoid single letters**. Which is easier to search? X or FirstName?

# Types of Classes

# Types of Classes

- Standard class
- Static class
- Nested class
- Abstract class
- Sealed class
- Partial class

# Nested Classes

- Nested classes are classes that are defined within the scope of another class.
- A nested type has access to all of the members that are accessible to its containing type.
- It can access private and protected members of the containing type, including any inherited protected members.

```
public class Container
{
    private class Nested
    {
        private Container parent;

        public Nested()
        {
        }
        public Nested(Container parent)
        {
            this.parent = parent;
        }
    }
}
```

# Abstract Classes

- An abstract class **cannot be instantiated**.
- Provides a base class implementation to be shared by multiple derived classes.
- Methods of an abstract class can be marked abstract too meaning the derived classes must implement the method.

```
public abstract class AbstractClass
{
    0 references
    public string Display()
    {
        return "Displayed.";
    }
    0 references
    public abstract string MakeItSo(int command);
}
```

# Sealed Classes

- Sealed classes prevent derivation – meaning, you can’t derive another class from a sealed class.
- Sealed classes can have a slight run-time performance gain because it can be guaranteed that there are no derived classes.
- You can also seal an overridden member of a derived class thereby removing the virtual aspect for any further derived classes.

# Partial Class

- A class that has the definition split up between **two or more files**

# User-defined structs

- A **struct** is a **value type** with similar qualities to a class except…
  - Structs can implement interfaces but NOT inherit from another struct
  - Cannot have protected members
- Suitable for representing lightweight objects.
  - Example: Point, Rectangle, Color
- A struct might be more efficient than a class. **Why?**
  - Value types don't need a reference variable and can be created without calling a constructor

```
struct Pos
{
    public int x, y;
}
0 references
static void Main(stri
{
    Pos pos;
    pos.x = 10;
    pos.y = 20;
```

# Static Classes

- Static classes are just like standard classes except that **they cannot be instantiated**. In other words, you can't new a static class.
  - `StaticClass sC = new StaticClass(); //not allowed!`
- Because you cannot create an instance of a static class, you can only gain access to its members by using the class name itself.
  - `int val = StaticClass.Value;`
- Good for grouping a set of functions that don't need internal fields.
- Also good for representing a type that you only need 1 instance of for the app.
- Remains in memory for the lifetime of the application.

# Factory Challenge

1. In the class library, create a static class called WeaponFactory.
2. Add a CreateWeapon method that returns a new FantasyWeapon.
   - NOTE: You will need to pass the same parameters to CreateWeapon that you pass to the FantasyWeapon constructor.
3. Call the CreateWeapon method from Main.

LINKS

Static classes

VIDEOS

# Derive from a class

# OOP: The Four Pillars

- The Four Pillars of OOP
  - Abstraction
  - Encapsulation
  - **Inheritance**
  - Polymorphism

# OOP: Inheritance

a way to

**REUSE, EXTEND, CHANGE**

the behavior of a class

# OOP: Inheritance

DERIVED BASE

```
class MetroBus : Bus
{
    int _faresCollected = 0;

    0 references
    public void AcceptFare(int fareAmount)
    {
        _faresCollected += fareAmount;
    }
}
```

# Constructors

# OOP: Inheritance

- Constructors:
  - When a derived class is created (ex. new DerivedClass()), the **base constructor is called *FIRST***, then the derived class's constructor is called.
  - Why? You want your base class to be fully initialized before the derived class gets initialized.
  - Something in your derived class might be dependent on the base class being initialized.

# OOP: Inheritance

- The derived class constructor ***must*** call a base class constructor ***IF*** the default constructor is not defined
- If the base class has multiple constructors, you can choose which constructor is appropriate to call.

# OOP: Inheritance

```
3 references
public class Spaceship
{
    1 reference
    public string Name { get; set; }
    1 reference
    public Spaceship(string name)
    {
        Name = name;
    }
}
```

Call the base constructor

```
public class Warship : Spaceship
{
    1 reference
    public float Speed { get; set; }
    0 references
    public Warship(string name, float speed) : base(name)
    {
        Speed = speed;
    }
}
```

# Inheritance Challenge

1. Create a class called BowWeapon that derives from FantasyWeapon
2. Add the following to BowWeapon:
   - Properties:
     - ArrowCapacity which should be an int
     - ArrowCount which should be an int
   - Constructors:
     - An overloaded constructor that initializes Arrow Capacity and Count

Example:

```
public Warship(string name, float speed) : base(name)
{
```

LINKS

Inheritance

VIDEOS

# Upcasting

# OOP: Upcasting

- **Upcasting**
  - It is always safe because the compiler knows if one type inherits from another.
  - From derived -> base

Derived to base

```
Warship enterprise = new Warship("Enterprise", 9.9F);
Spaceship federationShip = enterprise;
```

# Upcasting Challenge

1. Modify the Inventory class. Change items to be a List of FantasyWeapon.
2. Modify AddItem to take an FantasyWeapon parameter.
3. Modify main to pass different kinds of FantasyWeapon (FantasyWeapons, BowWeapons) to the AddItem method.

LINKS

VIDEOS

# Downcasting

# OOP: Downcasting

- **Downcasting**
  - It is NOT SAFE because the compiler does not know if one variable was actually created as a derived type.
  - From base -> derived

Base to derived (not allowed)

```
enterprise = federationShip;
```

# OOP: Downcasting

- How to downcast safely:
  - Use the as keyword.
  - Must *always* check for null.

```
Warship flagship = federationShip as Warship;
if(flagship != null)
{
    //do something with flagship
}
```

# OOP: Downcasting

- How to downcast safely:
  - Use pattern matching.

```
if(federationShip is Warship ship)
{
    //do something with ship
}
```

# Downcasting Challenge

1. Add a method called PrintInventory to the Inventory class.
2. In the method, loop over the inventory and print the property information: Rarity, level, max damage, and cost.
3. If the weapon is a BowWeapon, also print the arrow capacity and arrow count.
4. Call PrintInventory from Main.

Example:

```
if(federationShip is Warship ship)
{
```

LINKS

Pattern Matching

VIDEOS