

1. Inheritance

Definition: Inheritance is a fundamental OOP principle where a class (child/derived class) can inherit properties, methods, and fields from another class (parent/base class). This enables code reusability and establishes an "is-a" relationship.

In Your Code:

```
class child : parent // child inherits from parent
class subchild : child // subchild inherits from child (multi-level
inheritance)
```

Key Points:

- C# supports single inheritance (a class can inherit from only one base class)
- C# supports multi-level inheritance (child → parent, subchild → child → parent)
- The derived class inherits all accessible members (public, protected, internal) from the base class
- Private members are not directly accessible but exist in the derived class
- Inheritance creates a hierarchical relationship between classes

Benefits:

- Code reusability: Don't repeat code across similar classes
- Extensibility: Easy to add new features by extending existing classes
- Maintainability: Changes in base class automatically reflect in derived classes
- Polymorphism: Enables treating derived class objects as base class objects

2. Access Modifiers

Definition: Access modifiers control the visibility and accessibility of class members from other parts of the code.

Types in Your Code:

Public

```
public int x { get; set; }
public virtual void show()
```

- Accessible from anywhere (inside class, derived classes, other classes, other assemblies)
- No restrictions on access
- Use when you want to expose functionality to the outside world

Private (Default)

csharp

```
int z; // private by default
```

- Only accessible within the same class
- Not accessible in derived classes directly
- Use for internal implementation details that should be hidden
- Enforces encapsulation

Internal

```
internal class Program
```

- Accessible within the same assembly (project)
- Not accessible from other assemblies
- Use for classes/members that should be available within your project but not to external consumers

Encapsulation Benefit: Access modifiers help implement encapsulation by hiding internal implementation details and exposing only necessary interfaces.

3. Properties

Definition: Properties are members that provide a flexible mechanism to read, write, or compute the values of private fields. They use accessors (get and set).

In Your Code:

csharp

```
public int x { get; set; } // Auto-implemented property
public int y { get; set; }
public int a { get; set; }
```

Auto-Implemented Properties:

- The compiler automatically creates a private backing field
- Simplified syntax when no additional logic is needed
- Equivalent to:

csharp

```
private int _x;
public int x
{
    get { return _x; }
    set { _x = value; }
}
```

Benefits:

- Provides controlled access to fields
- Can add validation logic later without changing the interface
- Supports data binding in frameworks
- Can be read-only or write-only by omitting get or set
- Can have different access levels (e.g., public get, private set)

4. Constructors

Definition: A constructor is a special method that is called automatically when an object is created. It initializes the object's state.

Types in Your Code:

Default Constructor (Parameterless)

csharp

```
public parent()
{
    // Called when: new parent()
}
```

Parameterized Constructor

csharp

```
public parent(int z)
{
```

```
    this.z = z; // Initialize private field z
}
```

Constructor Chaining with `base()`

csharp

```
public child(int x, int y, int z) : base(z)
{
    this.x = x;
    this.y = y;
}
```

How `base()` Works:

- `:base(z)` calls the parent class constructor with parameter z
- Executes parent constructor BEFORE the child constructor body
- Ensures proper initialization of inherited members
- If not specified, the parameterless base constructor is called automatically

Execution Order:

1. subchild constructor called
2. Calls child constructor via `base(x,y,z)`
3. child constructor calls parent constructor via `base(z)`
4. parent(z) executes → initializes z
5. child constructor body executes → initializes x, y
6. subchild constructor body executes → initializes a

Important Rules:

- If base class has no parameterless constructor, derived class must explicitly call a parameterized base constructor
- Constructors are not inherited
- Constructor chaining ensures proper initialization hierarchy

5. Virtual and Override Keywords

Definition: These keywords enable polymorphism by allowing derived classes to provide their own implementation of base class methods.

Virtual Keyword

csharp

```
public virtual void show()
{
    Console.WriteLine($"x={x}, z={z}");
}
```

Purpose:

- Marks a method in the base class as overridable
- Provides a default implementation that can be replaced
- Enables runtime polymorphism (late binding)
- Without `virtual`, the method cannot be overridden (only hidden with `new`)

Override Keyword

csharp

```
public override void show()
{
    base.show(); // Call parent implementation
    Console.WriteLine($"y={y}");
}
```

Purpose:

- Provides a new implementation of a virtual method in the derived class
- Maintains the polymorphic chain
- Must match the signature of the virtual method exactly
- Can call the base implementation using `base.show()`

Polymorphic Behavior:

csharp

```
parent p = new child(1,2,3);
p.show(); // Calls child's override version, not parent's
```

Why This Matters:

- The actual type of the object (child) determines which method runs
- Not the reference type (parent)

- This is runtime polymorphism or dynamic binding
- Enables flexible, extensible code

6. New Keyword (Method Hiding)

Definition: The `new` keyword hides a base class member in the derived class instead of overriding it. This breaks the polymorphic chain.

In Your Code:

csharp

```
public new void show() // In subchild class
{
    base.show();
    Console.WriteLine($"a={a}");
}
```

Difference Between new and override:

With override (polymorphic):

csharp

```
parent p = new child(1,2,3);
p.show(); // Calls child.show() - polymorphic
```

With new (hiding):

csharp

```
parent p = new subchild(1,2,3,4);
p.show(); // Calls child.show(), NOT subchild.show()
// Because new breaks polymorphism

subchild s = new subchild(1,2,3,4);
s.show(); // Calls subchild.show() - direct reference
```

When to Use:

- Use `override` when you want polymorphic behavior (99% of cases)
- Use `new` when you intentionally want to hide a base member and break polymorphism
- `new` is rarely the right choice in good OOP design

Commented Code Example:

csharp

```
// public new string x { get; set; } // Hiding parent's int x with string x  
// private new int z; // Hiding parent's private z with child's private z
```

These would hide the parent's members but with different types, which can be confusing and is generally bad practice.

7. Polymorphism

Definition: Polymorphism means "many forms." It allows objects of different types to be treated uniformly through a common interface while exhibiting different behaviors.

In Your Code - The `display()` Method:

csharp

```
static void display(parent p)  
{  
    p.show(); // Calls appropriate show() based on actual object type  
}
```

Usage Examples:

csharp

```
display(new parent(1));           // Calls parent.show()  
display(new child(1,2,3));        // Calls child.show() – polymorphic  
display(new subchild(1,2,3,4));   // Calls child.show() (not subchild due to  
'new')
```

How It Works:

1. Method accepts a `parent` reference
2. You can pass any object that IS-A parent (`parent`, `child`, or `subchild`)
3. At runtime, the CLR determines the actual type of the object
4. Calls the most-derived `override` implementation

Benefits:

- Write flexible code that works with many types

- Add new derived classes without changing existing code
- Implement common interfaces for different implementations
- Foundation for design patterns and frameworks

Types of Polymorphism:

Compile-Time (Static) Polymorphism:

- Method overloading
- Operator overloading
- Resolved at compile time

Runtime (Dynamic) Polymorphism:

- Method overriding (your code demonstrates this)
- Resolved at runtime based on actual object type
- Requires virtual/override keywords

8. Sealed Keyword (Commented in Your Code)

csharp

```
// sealed class test : a
// {
// }
// class b : test // ERROR: Cannot inherit from sealed class
```

Definition: The `sealed` keyword prevents a class from being inherited or a method from being overridden further.

Sealed Class:

- Cannot be used as a base class
- No class can inherit from it
- Commonly used for utility classes or security reasons
- Example: `System.String` is sealed

Sealed Method:

csharp

```
public sealed override void show() // Cannot be overridden in derived classes
```

When to Use:

- When you want to prevent further inheritance for security/design reasons
- When you want to optimize performance (slight benefit, sealed methods can be inlined)
- When your class is complete and shouldn't be extended
- Example: Framework classes that shouldn't be modified

Benefits:

- Security: Prevents malicious code from extending your class
- Design integrity: Ensures your class behavior isn't altered
- Performance: Minor optimization opportunities

9. this Keyword

Definition: `this` is a reference to the current instance of the class. It's used to distinguish between class members and parameters with the same name.

In Your Code:

csharp

```
public parent(int z)
{
    this.z = z; // this.z is the field, z is the parameter
}

public child(int x, int y, int z) : base(z)
{
    this.x = x; // Distinguish property x from parameter x
    this.y = y;
}
```

Uses of this:

1. **Resolve naming conflicts** (most common):

csharp

```
this.x = x; // Assign parameter x to property x
```

2. **Call other constructors**:

csharp

```
public parent() : this(0) // Call parent(int z) constructor
{
}
```

3. Pass current instance to another method:

csharp

```
someMethod(this);
```

4. Return current instance for method chaining:

csharp

```
public parent SetX(int x)
{
    this.x = x;
    return this; // Enable: obj.SetX(5).SetY(10)
}
```

10. base Keyword

Definition: `base` is used to access members of the base class from within a derived class.

In Your Code:

Constructor Chaining:

csharp

```
public child(int x, int y, int z) : base(z)
// Calls parent constructor with parameter z
```

Calling Base Method:

csharp

```
public override void show()
{
    base.show(); // Call parent's show() method
    Console.WriteLine($"y={y}");
}
```

Important Uses:

1. **Call base constructor:** Initialize inherited members properly
2. **Call base method implementation:** Extend rather than replace base functionality
3. **Access hidden members:** If you've used `new`, you can still access the base version

Example Flow:

csharp

```
subchild s = new subchild(1,2,3,4);
s.show();

// Execution:
// 1. subchild.show() starts
// 2. Calls base.show() → child.show()
// 3. child.show() calls base.show() → parent.show()
// 4. parent.show() prints x and z
// 5. Returns to child.show(), prints y
// 6. Returns to subchild.show(), prints a
```

11. Upcasting and Downcasting (Implicit in Your Code)

Upcasting (Implicit)

csharp

```
parent p = new child(1,2,3); // Automatic, safe
parent p2 = new subchild(1,2,3,4); // Automatic, safe
```

Definition: Converting a derived class reference to a base class reference.

Characteristics:

- Always safe (every child IS-A parent)
- Implicit (no cast operator needed)
- May lose access to derived class members
- Enables polymorphism

Downcasting (Explicit - Not in Your Code)

csharp

```

parent p = new child(1,2,3);
child c = (child)p; // Explicit cast required, can fail at runtime

// Safe downcasting:
if (p is child c2) // Pattern matching (C# 7+)
{
    c2.y = 10; // Safe to use
}

// Or:
child c3 = p as child; // Returns null if cast fails
if (c3 != null)
{
    c3.y = 10;
}

```

Characteristics:

- Requires explicit cast
- Can fail at runtime with InvalidCastException
- Should use `is` or `as` operators for safety
- Regains access to derived class members

12. Method Execution with Polymorphism - Detailed Analysis

Let's trace the execution of your final line:

csharp

```
display(new subchild(1,2,3,4));
```

Step-by-Step:

1. Object Creation:

- `new subchild(1,2,3,4)` creates a subchild object
- Constructor chain: subchild → child → parent
- Memory allocated for all fields (z, x, y, a)

2. Upcasting:

- subchild reference implicitly converted to parent reference
- Parameter: `parent p = new subchild(1,2,3,4)`

3. Method Call:

- `p.show()` is called

- CLR checks the actual object type: subchild
- Looks for show() implementation in subchild

4. Method Resolution:

- subchild.show() uses new keyword (hiding, not overriding)
- Since p is parent reference, polymorphism goes to child.show()
- child.show() is an override, so it's called

5. Execution:

- child.show() executes base.show() → parent.show() runs
- Prints: x=1, z=3
- child.show() continues: prints y=2
- subchild.show() is NOT called because of new keyword

If subchild.show() used override instead of new:

csharp

```
public override void show() // Instead of new
{
    base.show();
    Console.WriteLine($"a={a}");
}
```

```

Then output would be:

```
```
x=1, z=3
y=2
a=4

```

Key Takeaways for OOP Design:

- **Prefer override over new:** Maintains polymorphic behavior
- **Use virtual for extensibility:** Mark methods virtual if derived classes might need to customize them
- **Constructor chaining is essential:** Use base() to ensure proper initialization
- **Access modifiers enforce encapsulation:** Hide implementation details, expose only necessary interfaces
- **Polymorphism enables flexible code:** Write methods that work with base classes, automatically work with derived classes
- **Properties over public fields:** Use properties for better encapsulation and future flexibility

By Abdullah Ali

Contact : +201012613453