Self-Study

1. Casting Operator

Casting in C#

Casting in C# refers to converting an expression of one type to another. There are several types of casts:

1. **Built-in Casts**:
    - **Numeric Casts**: Implicit (e.g., int to long) or explicit (e.g., double to int), handled by the CLR with IL instructions like conv.i4 or conv.r8.
    - **Reference Type Casts**: Upcasting (e.g., Derived to Base) or downcasting (e.g., Base to Derived), managed by the CLR with isinst for type checking.
    - **Boxing/Unboxing**: Converting value types to/from object or interfaces, using box and unbox.any IL instructions.
    - These casts are built into the language and CLR and do not involve operator overloading. They are implemented as low-level IL instructions, not as user-defined methods.
2. **User-Defined Casts**:
    - These are custom conversions defined by a class or struct using the implicit or explicit operator keywords.
    - User-defined casts are a form of operator overloading, as they allow developers to define how one type can be converted to another by overloading the casting operation.

User-Defined Casting Operators as Operator Overloading

In C#, operator overloading allows developers to define custom behavior for operators (e.g., +, -, ==, or casts) for user-defined types. The casting

operator is overloaded by defining implicit or explicit conversion operators in a class or struct.

Syntax for User-Defined Casting Operators

```
public class MyNumber
{
   private int value;
   public MyNumber(int v) => value = v;

   // Implicit conversion from MyNumber to int
   public static implicit operator int(MyNumber n) => n.value;

   // Explicit conversion from int to MyNumber
   public static explicit operator MyNumber(int i) => new
MyNumber(i);
}
```

Usage

```
MyNumber num = new MyNumber(42);
int i = num; // Implicit cast (no explicit syntax needed)
MyNumber num2 = (MyNumber)42; // Explicit cast (requires (MyNumber))
```

How It's Operator Overloading

- **Operator Keyword**: The implicit and explicit keywords are used to define conversion operators, similar to how operator + defines addition. The casting operator is essentially a special operator that handles type conversion.
- **Method Call**: Under the hood, the C# compiler translates a user-defined cast into a call to the static method defined by the implicit or explicit operator. For example:

```
int i = num; // Compiler calls: int i = MyNumber.op_Implicit(num);
```

- IL code (simplified):

```
ldloc.0          // Load MyNumber instance

call int32 MyNumber::op_Implicit(MyNumber)

stloc.1          // Store result in i
```

- **Custom Logic**: Like other overloaded operators (e.g., + or ==), user-defined casting operators allow custom logic to define how one type is converted to another, making them a clear instance of operator overloading.

Key Characteristics

- **Static Methods**: Conversion operators are static methods named op_Implicit or op_Explicit in the IL, following the same naming convention as other overloaded operators (e.g., op_Addition for +).
- **Type Safety**: The compiler ensures that implicit operators are used only for safe conversions (no data loss), while explicit operators require explicit cast syntax for potentially unsafe conversions.
- **Single Direction**: Each operator defines a conversion in one direction (e.g., MyNumber to int or int to MyNumber). Multiple operators can be defined for different conversions.

2. Upcasting Vs Downcasting

In C#, **upcasting** and **downcasting** are casting operations used with reference types in an inheritance hierarchy, converting between base and derived classes. Each has distinct characteristics, use cases, and implementation details. Below is a detailed comparison of upcasting vs. downcasting, with the key differences section presented as bullet points, as requested.

Definitions

- **Upcasting**:
  - Converting a derived class reference to a base class reference (e.g., Derived to Base).
  - Example: Base b = new Derived();
  - Moves "up" the inheritance hierarchy (from a more specific type to a less specific type).
  - Always safe and implicit (no explicit cast syntax required).
- **Downcasting**:
  - Converting a base class reference to a derived class reference (e.g., Base to Derived).
  - Example: Derived d = (Derived)b;
  - Moves "down" the inheritance hierarchy (from a less specific type to a more specific type).
  - Potentially unsafe, requires explicit cast syntax, and may throw an InvalidCastException at runtime if the object is not of the target type.

Key Differences

- **Direction**: Upcasting moves from a derived class to a base class (up the inheritance hierarchy), while downcasting moves from a base class to a derived class (down the hierarchy).
- **Syntax**: Upcasting is implicit, requiring no cast operator (e.g., Base b = new Derived();), whereas downcasting requires an explicit cast (e.g., Derived d = (Derived)b;).
- **Safety**: Upcasting is always safe because a derived class is guaranteed to be a valid instance of its base class, but downcasting is potentially unsafe and may fail, throwing an InvalidCastException if the object's runtime type is incompatible.
- **Purpose**: Upcasting generalizes a type for polymorphic behavior or to use base class features, while downcasting specializes a type to access derived class-specific members.

- **Runtime Check**: Upcasting requires no runtime type checking, as it’s guaranteed to succeed, whereas downcasting involves a runtime type check using the isinst IL instruction to verify type compatibility.
- **Performance**: Upcasting is faster, with no runtime overhead, while downcasting is slower due to the runtime type-checking process.
- **Use Case**: Upcasting is used when passing derived objects to methods expecting base types or for storing objects in collections, whereas downcasting is used to access derived class-specific methods or properties from a base class reference.