**Differences Between Interfaces and Abstract Classes**

Before we explore scenarios, it's essential to understand the fundamental differences:

1. **Implementation**:
   * **Interfaces**: Can contain abstract methods (without implementation) and, starting with C# 8.0, **default implementations** (methods with a body). However, they cannot hold state (fields).
   * **Abstract Classes**: Can contain both abstract methods and fully implemented methods. They can also hold state through fields and properties.
2. **Inheritance**:
   * **Interfaces**: Support multiple inheritance. A class can implement multiple interfaces.
   * **Abstract Classes**: Do not support multiple inheritance. A class can inherit from only one abstract (or concrete) class.
3. **Accessibility Modifiers**:
   * **Interfaces**: Members are implicitly public. Starting with C# 8.0, interfaces can have more granular access modifiers for default implementations.
   * **Abstract Classes**: Members can have any access modifier (public, protected, private, etc.).
4. **Constructors**:
   * **Interfaces**: Cannot have constructors.
   * **Abstract Classes**: Can have constructors, which can be invoked by derived classes.

**When to Use Interfaces**

**1. Defining Contracts Without Shared Code or State**

**Scenario**: You have different classes that need to adhere to a common set of functionalities but do not share any implementation details or state.

**Example**: Payment Processing in an E-commerce System.

* **Interface Definition**:

public interface IPaymentProcessor

{

void ProcessPayment(decimal amount);

bool RefundPayment(decimal amount);

}

**Implementations:**

public class PayPalProcessor : IPaymentProcessor

{

public void ProcessPayment(decimal amount)

{

// PayPal-specific payment processing

}

public bool RefundPayment(decimal amount)

{

// PayPal-specific refund processing

return true;

}

}

public class StripeProcessor : IPaymentProcessor

{

public void ProcessPayment(decimal amount)

{

// Stripe-specific payment processing

}

public bool RefundPayment(decimal amount)

{

// Stripe-specific refund processing

return true;

}

}

**Why Interface?**

* **Flexibility**: Multiple payment processors can be implemented and used interchangeably.
* **No Shared Code Needed**: Each processor has its unique implementation without needing shared state or behavior.

**2. Enabling Multiple Inheritance of Type**

**Scenario**: A class needs to inherit behaviors from multiple sources.

**Example**: A Smart Device that is both a Camera and a GPS device.

* **Interface Definitions**:

public interface ICamera

{

void TakePhoto();

}

public interface IGPS

{

void GetLocation();

}

**Implementation:**

public class SmartDevice : ICamera, IGPS

{

public void TakePhoto()

{

// Implementation for taking photo

}

public void GetLocation()

{

// Implementation for getting location

}

}

**Why Interface?**

* **Multiple Capabilities**: SmartDevice can be both a camera and a GPS without being limited by single inheritance constraints.

**Example**: Logging Mechanism.

* **Interface with Default Method**:

public interface ILogger

{

void Log(string message);

// Default implementation

void LogError(string message)

{

Log($"ERROR: {message}");

}

}

**Implementation:**

public class ConsoleLogger : ILogger

{

public void Log(string message)

{

Console.WriteLine(message);

}

// Inherits default LogError implementation

}

public class FileLogger : ILogger

{

public void Log(string message)

{

// Write message to a file

}

public void LogError(string message)

{

// Custom error logging for files

Log($"FILE ERROR: {message}");

}

}

**When to Use Abstract Classes**

**1. Sharing Common Code and State**

**Scenario**: Multiple related classes share some common implementation or state that you want to centralize.

**Example**: Employee Management System.

* **Abstract Class Definition**:

public abstract class Employee

{

public string Name { get; set; }

public string EmployeeID { get; set; }

protected double Salary { get; set; }

public Employee(string name, string employeeID, double salary)

{

Name = name;

EmployeeID = employeeID;

Salary = salary;

}

// Abstract method

public abstract void CalculateBonus();

// Concrete method

public void DisplayEmployeeInfo()

{

Console.WriteLine($"Name: {Name}, ID: {EmployeeID}, Salary: {Salary}");

}

}

**Derived classes:**

public class Manager : Employee

{

public Manager(string name, string employeeID, double salary)

: base(name, employeeID, salary) { }

public override void CalculateBonus()

{

double bonus = Salary \* 0.20;

Console.WriteLine($"Manager Bonus: {bonus}");

}

}

public class Developer : Employee

{

public Developer(string name, string employeeID, double salary)

: base(name, employeeID, salary) { }

public override void CalculateBonus()

{

double bonus = Salary \* 0.10;

Console.WriteLine($"Developer Bonus: {bonus}");

}

}

**Why Abstract Class?**

* **Shared Implementation**: DisplayEmployeeInfo method is shared among all employees.
* **Shared State**: Common properties like Name, EmployeeID, and Salary are centralized.
* **Enforcing a Base Structure**: All employees must implement CalculateBonus, ensuring consistency.

**2. Providing Base Functionality with Some Abstract Methods**

**Scenario**: You want to provide a foundational behavior that can be extended or customized by derived classes.

**Example**: Notification System.

* **Abstract Class Definition**:

public abstract class Notifier

{

public string Recipient { get; set; }

public Notifier(string recipient)

{

Recipient = recipient;

}

// Abstract method to send notification

public abstract void Send(string message);

// Concrete method with shared functionality

public void LogNotification(string message)

{

// Log the notification details

Console.WriteLine($"Notification to {Recipient}: {message}");

}

}

**Derived Class:**

public class EmailNotifier : Notifier

{

public EmailNotifier(string recipient) : base(recipient) { }

public override void Send(string message)

{

// Email sending logic

Console.WriteLine($"Email sent to {Recipient}: {message}");

LogNotification(message);

}

}

public class SmsNotifier : Notifier

{

public SmsNotifier(string recipient) : base(recipient) { }

public override void Send(string message)

{

// SMS sending logic

Console.WriteLine($"SMS sent to {Recipient}: {message}");

LogNotification(message);

}

}

**Why Abstract Class?**

* **Shared Functionality**: LogNotification method is used by all notifiers.
* **Enforced Customization**: Each notifier must implement its own Send method.

**3. When Constructors and Initialization Are Needed**

**Scenario**: You need to initialize some common state or enforce certain setup procedures for all derived classes.

**Example**: Game Characters in a Game Engine.

* **Abstract Class Definition**:

public abstract class GameCharacter

{

public string Name { get; private set; }

public int Health { get; protected set; }

public GameCharacter(string name, int health)

{

Name = name;

Health = health;

}

// Abstract method for attacking

public abstract void Attack(GameCharacter target);

// Concrete method for taking damage

public void TakeDamage(int damage)

{

Health -= damage;

Console.WriteLine($"{Name} takes {damage} damage. Remaining Health: {Health}");

}

}

**Derived Class:**

public class Warrior : GameCharacter

{

public Warrior(string name, int health) : base(name, health) { }

public override void Attack(GameCharacter target)

{

int damage = 15;

Console.WriteLine($"{Name} swings a sword at {target.Name} for {damage} damage.");

target.TakeDamage(damage);

}

}

public class Mage : GameCharacter

{

public Mage(string name, int health) : base(name, health) { }

public override void Attack(GameCharacter target)

{

int damage = 25;

Console.WriteLine($"{Name} casts a fireball at {target.Name} for {damage} damage.");

target.TakeDamage(damage);

}

}

**Why Abstract Class?**

* **Initialization Logic**: Ensures all characters have a Name and Health upon creation.
* **Shared Behavior**: TakeDamage method is common to all characters.

**Comparing Scenarios: When to Use Interface vs. Abstract Class**

Let's explore some real-time scenarios where you might choose one over the other:

**1. Extending Functionality Across Unrelated Classes**

**Scenario**: You want to add logging capabilities to various unrelated classes, such as Order, Customer, and Product.

* **Solution**: Use an **interface** like ILogger that these classes can implement.

**Why Interface?**

* **No Shared Code Needed**: These classes don’t share a common ancestor but can all implement logging.
* **Multiple Implementations**: Classes can implement multiple interfaces if needed.

**2. Defining a Template with Shared Code and Enforced Methods**

**Scenario**: You’re building a GUI framework where all UI components share some common functionality (e.g., rendering, resizing) but also have unique behaviors.

* **Solution**: Use an **abstract class** like UIComponent that provides shared methods and enforces the implementation of unique behaviors.

**Why Abstract Class?**

* **Shared Code**: Common methods like Render() and Resize() can be implemented once.
* **State Management**: Shared properties like Width, Height, and Position can be maintained in the base class.

**3. Creating Plug-in Systems with Extensible Features**

**Scenario**: Developing a media player that can support various media formats through plugins.

* **Solution**: Define an **interface** like IMediaPlugin that all plugins must implement.

**Why Interface?**

* **Flexibility**: Plugins can come from various sources and need not share a common base class.
* **Multiple Plugins**: A media player might support multiple plugins simultaneously, benefiting from interface-based multiple inheritance.

**4. Enforcing a Base Structure with Some Shared Implementation**

**Scenario**: In a content management system, all content types (e.g., Article, Video, Image) share some common properties and behaviors but also have unique attributes.

* **Solution**: Use an **abstract class** like Content to define common properties (e.g., Title, Author) and enforce methods like Publish().

**Why Abstract Class?**

* **Shared Properties and Methods**: Avoid duplication by defining them in the abstract base.
* **Enforcement**: Ensure all content types implement essential behaviors.

**5. Providing Multiple Capabilities to a Single Class**

**Scenario**: A class Smartphone needs to support both telephony and internet capabilities.

* **Solution**: Implement multiple **interfaces** like ITelephony and IInternet.

**Why Interface?**

* **Multiple Inheritance**: C# allows a class to implement multiple interfaces, enabling it to support diverse capabilities without being restricted by single inheritance.

**Combined Usage: Interfaces and Abstract Classes Together**

In many real-world applications, interfaces and abstract classes are used together to leverage their respective strengths.

**Example**: Vehicle Management System.

* **Interface Definition**:

public interface IVehicle

{

void StartEngine();

void StopEngine();

}

**Abstract class definition:**

public abstract class VehicleBase : IVehicle

{

public string Make { get; set; }

public string Model { get; set; }

protected VehicleBase(string make, string model)

{

Make = make;

Model = model;

}

// Implementing interface methods with shared behavior

public virtual void StartEngine()

{

Console.WriteLine($"{Make} {Model} engine started.");

}

public virtual void StopEngine()

{

Console.WriteLine($"{Make} {Model} engine stopped.");

}

// Abstract method for unique behavior

public abstract void Drive();

}

**Concrete class Implementation:**

public class Car : VehicleBase

{

public Car(string make, string model) : base(make, model) { }

public override void Drive()

{

Console.WriteLine($"{Make} {Model} is driving on the road.");

}

}

public class Boat : VehicleBase

{

public Boat(string make, string model) : base(make, model) { }

public override void Drive()

{

Console.WriteLine($"{Make} {Model} is sailing on the water.");

}

// Overriding StartEngine for boat-specific behavior

public override void StartEngine()

{

Console.WriteLine($"{Make} {Model} boat engine started with a roar.");

}

}

**Why Combined Usage?**

* **Interface for Contract**: IVehicle ensures that all vehicles have StartEngine and StopEngine methods.
* **Abstract Class for Shared Implementation**: VehicleBase provides common properties and default behaviors, reducing duplication.
* **Flexibility in Concrete Classes**: Specific vehicles like Car and Boat can have unique implementations and override shared behaviors as needed.
* **Summary of When to Use Interfaces vs. Abstract Classes**

| **Criteria** | **Interface** | **Abstract Class** |
| --- | --- | --- |
| **Multiple Inheritance** | Supported (multiple interfaces can be implemented) | Not supported (single inheritance) |
| **Shared Code or State** | No (interfaces cannot hold state) | Yes (can hold fields and properties) |
| **Default Implementations** | Yes (since C# 8.0) | Yes |
| **Constructors** | No constructors | Yes constructors |
| **Versioning (Adding Methods)** | Easier with default implementations | Can lead to breaking changes if abstract methods are added |
| **Use Case Examples** | - Plug-in architectures | - Template method patterns |
|  | - Defining contracts for unrelated classes | - Shared base functionality |
|  | - Enabling multiple capabilities to a class | - Maintaining shared state |
| **When to Prefer Interface** | - When defining a contract without shared code | - When multiple unrelated classes need to implement the same functionality |
|  | - When multiple inheritance of type is required |  |
|  | - When you expect unrelated classes to implement your interface |  |
| **When to Prefer Abstract Class** | - When you have shared code and state to maintain | - When you need to provide common base functionality with some implementation |
|  | - When you want to define a base class with default behaviors | - When you want to enforce a base structure while allowing flexibility |
|  |  |  |

**1. Payment Processing System**

* **Interfaces for Different Payment Methods**:

public interface IPaymentMethod

{

void Pay(decimal amount);

void Refund(decimal amount);

}

public class CreditCardPayment : IPaymentMethod

{

public void Pay(decimal amount) { /\* Credit Card Payment Logic \*/ }

public void Refund(decimal amount) { /\* Credit Card Refund Logic \*/ }

}

public class PayPalPayment : IPaymentMethod

{

public void Pay(decimal amount) { /\* PayPal Payment Logic \*/ }

public void Refund(decimal amount) { /\* PayPal Refund Logic \*/ }

}

**Abstract Class for Common Transaction Logging:**

public abstract class TransactionLogger

{

public void LogTransaction(string transactionDetails)

{

// Common logging logic

Console.WriteLine($"Transaction Logged: {transactionDetails}");

}

public abstract void LogError(string errorDetails);

}

public class FileTransactionLogger : TransactionLogger

{

public override void LogError(string errorDetails)

{

// File-specific error logging

Console.WriteLine($"File Log Error: {errorDetails}");

}

}

public class DatabaseTransactionLogger : TransactionLogger

{

public override void LogError(string errorDetails)

{

// Database-specific error logging

Console.WriteLine($"Database Log Error: {errorDetails}");

}

}

**When to use Interface**

**Scenario,**  
Consider we want to start a service like "makemytrip.com" or "expedia.com", where we are responsible for displaying the flights from various flight service company and place an order from customer.   
Let’s keep our service as simple as,

1. Displaying flights available from vendors like "airasia", "british airways" and "emirates".
2. Place and order for seat to respective vendor.

***How should we design our application considering interfaces and abstract class? In this scenario, interface is useful or abstract class?***  
  
Remember, In this application, we don't own any flight. we are just a middle man/aggregator and our task is to first enquire "airasia", then enquire "british airways" and at last enquire "emirates" about the list of flights available and later if customer opts for booking then inform the respective flight vendor to do booking.  
For this, first we need to tell "airasia", "british airways" and "emirates" to give us list of flights, internally how they are giving the list that we don't care.

1. **This means I only care for method "getAllAvailableFlights()"**  
     
   "getAllAvailableFlights()" from "airasia" may have used SOAP service to return list of flights.  
   "getAllAvailableFlights()" from "british airways" may have used REST service to return list of flights.  
   "getAllAvailableFlights()" from "emirates" may have used CORBA service to return list of flights.  
     
   but we don't care how it is internally implemented and what we care is the contract method "**getAllAvailableFlights**" that all the flight vendor should provide and return list of flights.
2. Similarly, for booking I only care for method "**booking()**" that all vendors should have, internally how this vendors are doing booking that I don't care.

**To conclude: We know contract.**  
So we can say that we know the contract that irrespective of who the Flight vendor is, we need "**getAllAvailableFlights()**" and "**booking()**" method from them to run our aggregator service.

*In this situation, Interface is useful because we are not aware of the implementation of all the 2 methods required, and what we know is the contract methods that vendor(implementer) should provide. so due to this total abstraction and for defining the contract, interface is useful in this place.*

**Technically, we need to design our interface somewhat like below,**  
  


**When to use Abstract class**

**Scenario,**  
Consider we want to start a service like Bulk SMS sender, where we take orders from various telecom vendors like Airtel, France Telecom, Vodafone etc.

For this, we don't have to setup our own infrastructure for sending SMS like Mobile towers but we need to take care of government rules like after 9PM, we should not send promotional SMS, we should also not send SMS to users registered under Do Not Disturb(DND) service etc. Remember, we need to take care of government rules for all the countries where we are sending SMS.  
  
***Note:****for infrastructure like towers, we will be relying on vendor who is going to give us order.*  
*Example, In case of,*  
*Vodafone request us for bulk messaging, in that case we will use Vodafine towers to send SMS.*  
*Airtel request us for bulk messaging, in that case we will use Airtel towers to send SMS.*  
*What our job is to manage Telecom Regulations for different countries where we are sending SMS.*So what all methods we require would be somewhat like below,

[?](https://javabypatel.blogspot.com/2017/07/real-time-example-of-abstract-class-and-interface-in-java.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | **public** **void** eastablishConnectionWithYourTower(){     //connect using vendor way.     //we don't know how, candidate for abstract method  }    **public** **void** sendSMS(){     eastablishConnectionWithYourTower();     checkForDND();     checkForTelecomRules();     //sending SMS to numbers...numbers.     destroyConnectionWithYourTower()  }    **public** **void** destroyConnectionWithYourTower(){     //disconnect using vendor way.     //we don't know how, candidate for abstract method  }    **public** **void** checkForDND(){     //check for number present in DND.  }    **public** **void** checkForTelecomRules(){     //Check for telecom rules.  } |

Out of above 5 methods,

1. Methods we know is "sendSMS()", "checkForDND()", "checkForTelecomRules()".
2. Methods we don't know is "eastablishConnectionWithYourTower()", "destroyConnectionWithYourTower()".

we know how to check government rules for sending SMS as that is what our job is but  
we don't how to eastablish connection with tower and how to destroy connection with tower because this is purely customer specific, airtel has its own way, vodafone has its own way etc.  
  
*So in the given scenario, we know some methods but there also exist some methods which are unknown and depends on customers.*  
  
In this case, what will be helpful, abstarct class or interface?

***In this case, Abstract class will be helpful, because you know partial things like "checkForDND()", "checkForTelecomRules()" for sending sms to users but we don't know how to eastablishConnectionWithTower() and destroyConnectionWithTower() and need to depend on vendor specific way to connect and destroy connection from their towers.***



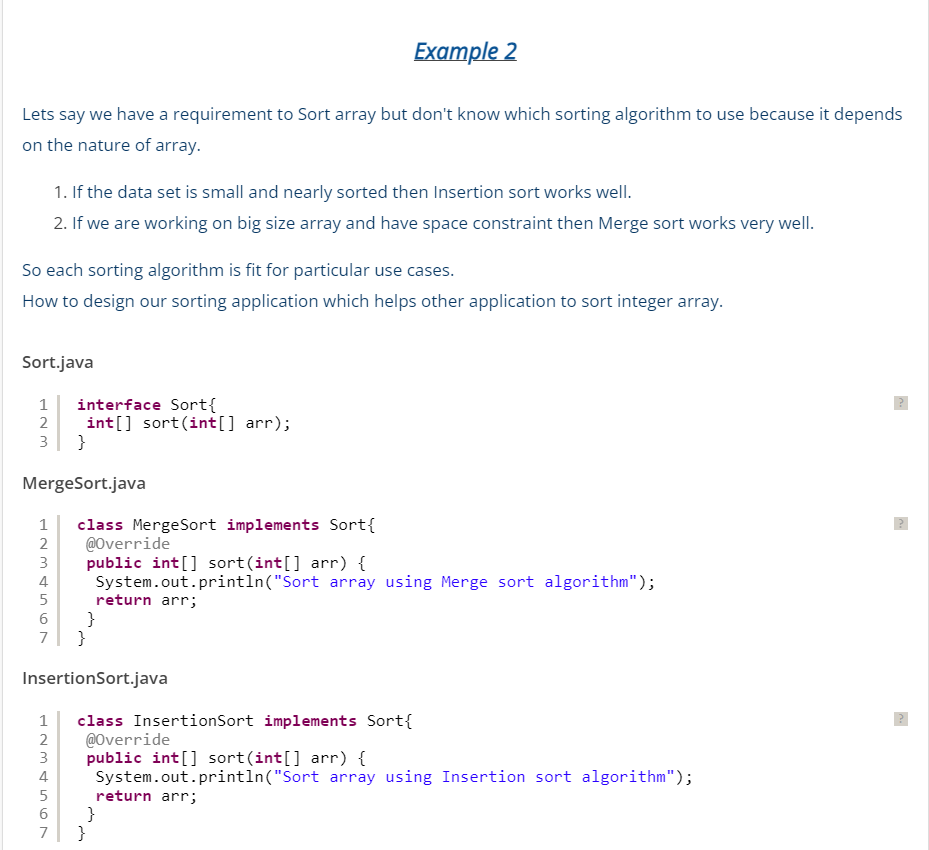
**So to summarize,**

**For Interface:**

*Interface is used when you don't know anything about implementation but know the contract that implementer should have to accomplish the task.*

**For Abstract class:**

*Abstract class is used when you know partial implementation, where say out of 5 methods, you know implementation of 3 methods and don't know implemenatation of 2 methods in that case 2 methods will be abstract and you need to rely on implementer as a contract to must provide body of abstract methods to accomplish the task.*



Example 3:

