Object-Oriented Programming in C#

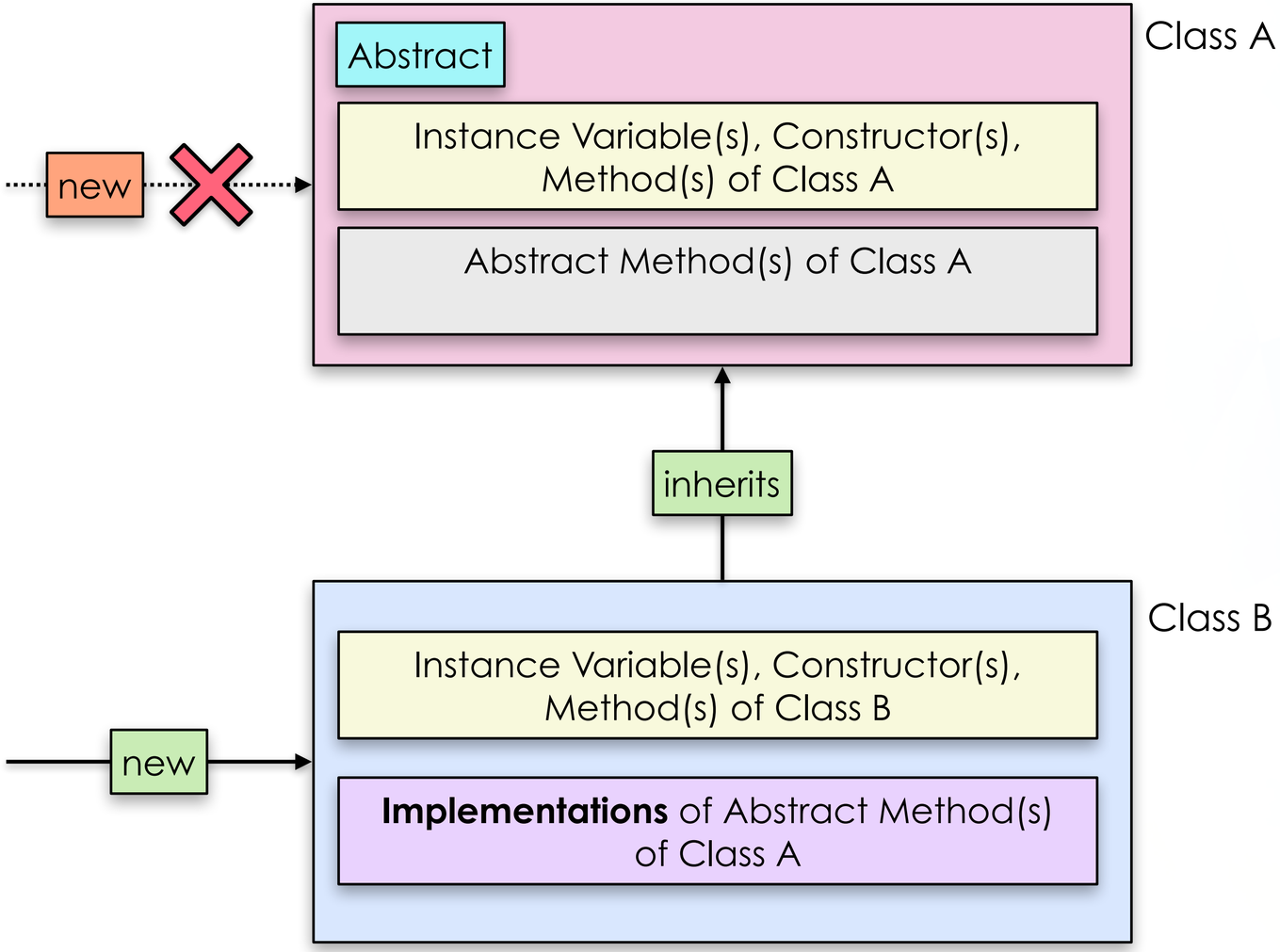
Tan Cher Wah
cherwah@nus.edu.sg

Abstract Class

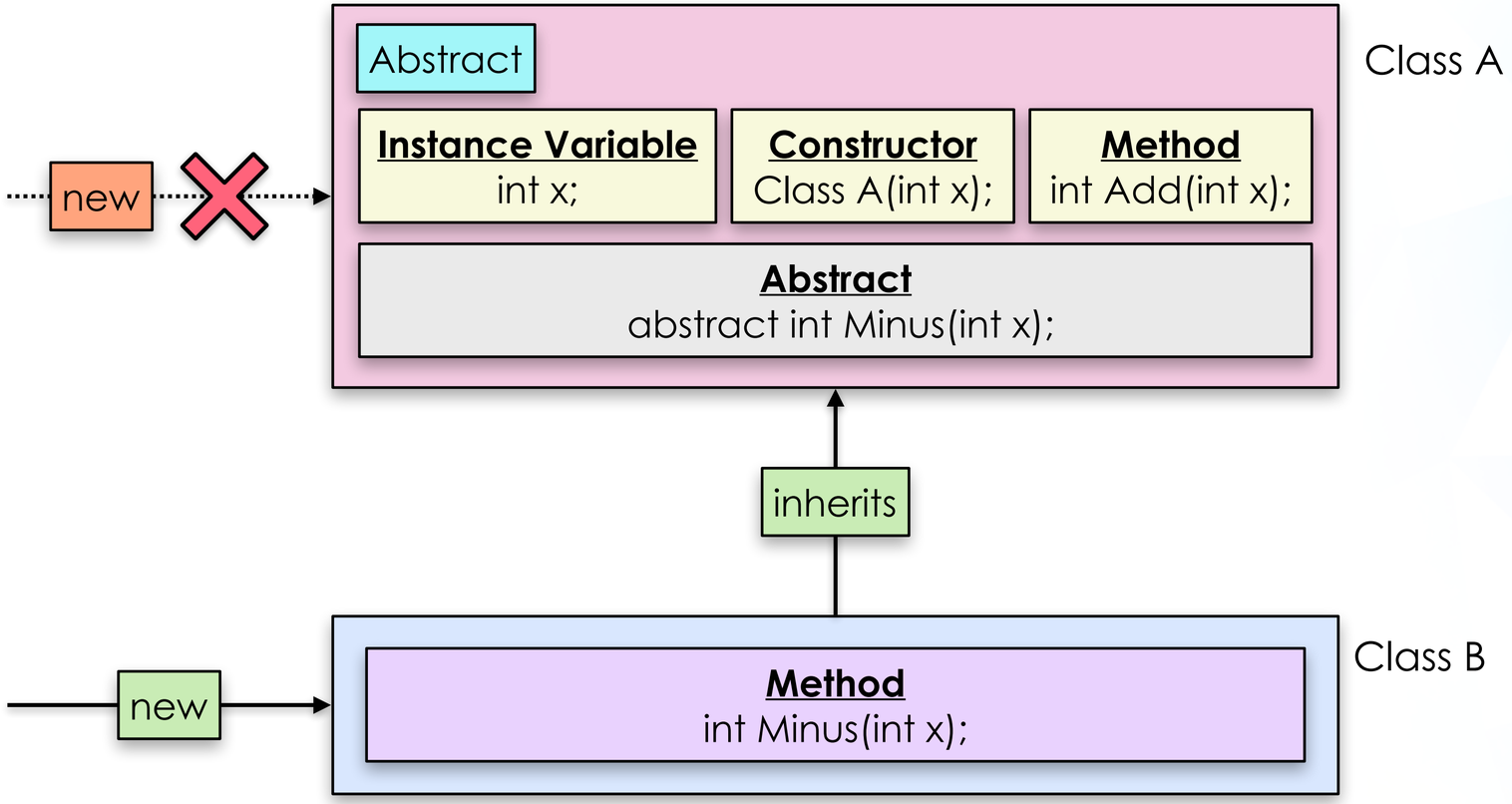
Abstract Class

- An Abstract Class is declared via the **abstract** keyword
- An Abstract Class can contain **instance variable(s)**, **constructor(s)**, **implemented method(s)** and **abstract method(s)** (not implemented methods)
- An Abstract Class **cannot be instantiated directly** and is designed to be inherited
- A child that inherits an Abstract Class must implement **all the abstract methods** defined by the Abstract Class
- A child can only inherit from a single Abstract Class

Abstract Class



Sample Abstract Class



Sample Abstract Class

How a sample C# Abstract Class looks like

```
public abstract class ClassA
{
    protected int x;

    public ClassA(int x) {
        this.x = x;
    }

    // zero or more abstract methods
    public int Add(int x) {
        // logic is implemented
        this.x += x;
        return this.x;
    }

    // one or more abstract methods (no logic)
    public abstract int Minus(int x);
}
```

Abstract Class

Subclass of Abstract Class implements all inherited abstract methods

```
public class ClassB : ClassA
{
    public ClassB(int x) : base(x) {
    }

    // implemented parent's abstract method
    public override int Minus(int x) {
        this.x -= x;
        return this.x;
    }
}
```

Abstract Class

The subclass of the Abstract Class can now be instantiated for use

```
ClassA a = new ClassB(10);

int x = a.Add(2);
Console.WriteLine("x = " + x);

x = a.Minus(1);
Console.WriteLine("x = " + x);
```

Output

```
x = 12
x = 11
```

Example: Bank Accounts

- Our software architect requires classes that manage bank accounts
- For a start, he needs two types of bank accounts – Savings Account and Checking Account.
- Both types of accounts need to have an Account Number and maintains a Balance (e.g. the amount of money in the account)
- Both types of accounts must allow money to be deposited and withdrawn
- A Savings Account receives a daily interest rate, but not a Checking Account
- Checking Account can transfer money to both Savings and Checking accounts, but Savings Account cannot do money transfer
- Let's see how he enforces his design decisions on his colleagues. ☺

BankAccount Abstract Class

```
public abstract class BankAccount
{
    protected string accountNo;
    protected float balance;

    public BankAccount(string accountNo, float balance) {
        this.accountNo = accountNo;
        this.balance = balance;
    }

    public float GetBalance() {
        return balance;
    }

    public void Deposit(float amt) {
        if (amt > 0) {
            balance += amt;
        }
    }

    public bool Withdraw(float amt) {
        if (amt <= 0 || balance < amt) {
            return false;
        }

        balance -= amt;
        return true;
    }

    public abstract float GetDailyInterest();
    public abstract bool TransferToAccount(
        BankAccount ba, float amt);
}
```

An abstract class can have **attributes**

An abstract class can contain **implemented methods**

The abstract class can have a **constructor**

These are **abstract methods** (no implementations)

SavingsAccount

SavingsAccount implemented **all abstract methods** of BankAccount

```
public class SavingsAccount : BankAccount
{
    protected float interest;

    public SavingsAccount(string accountNo, float balance)
    : base(accountNo, balance) {
        interest = 0.03f;   // 3% p.a.
    }

    public override float GetDailyInterest() {
        return (interest * balance) / 365;
    }

    public override bool TransferToAccount(BankAccount ba, float amt) {
        // no transfer allowed
        return false;
    }
}
```

Implemented the **two abstract methods** defined by parent

CheckingAccount

CheckingAccount implemented **all abstract methods** of BankAccount

```
public class CheckingAccount : BankAccount
{
    public CheckingAccount(string accountNo, float balance)
        : base(accountNo, balance)
    {
    }

    public override float GetDailyInterest() {
         // no daily interest
        return 0;
    }
```

Implemented abstract methods

```
    public override bool TransferToAccount(BankAccount ba, float amt) {
        if (Withdraw(amt)) {
            ba.Deposit(amt);
            return true;
        }

        return false;
    }
}
```

Using Savings and Checking Accounts

```
BankAccount sa = new SavingsAccount("SA-001", 100f);
BankAccount ca = new CheckingAccount("CA-001", 500f);

sa.Withdraw(50f);

ca.Deposit(100f);
ca.TransferToAccount(sa, 100f);

Console.WriteLine("Savings Balance: $" + sa.GetBalance());
Console.WriteLine("Checking Balance: $" + ca.GetBalance());
Console.WriteLine("Daily Interest: $" + ca.GetDailyInterest());
```

Child classes can be **instantiated** as they have implemented all of parents' abstract methods

Implemented methods **conform** to designs of abstract class

Output

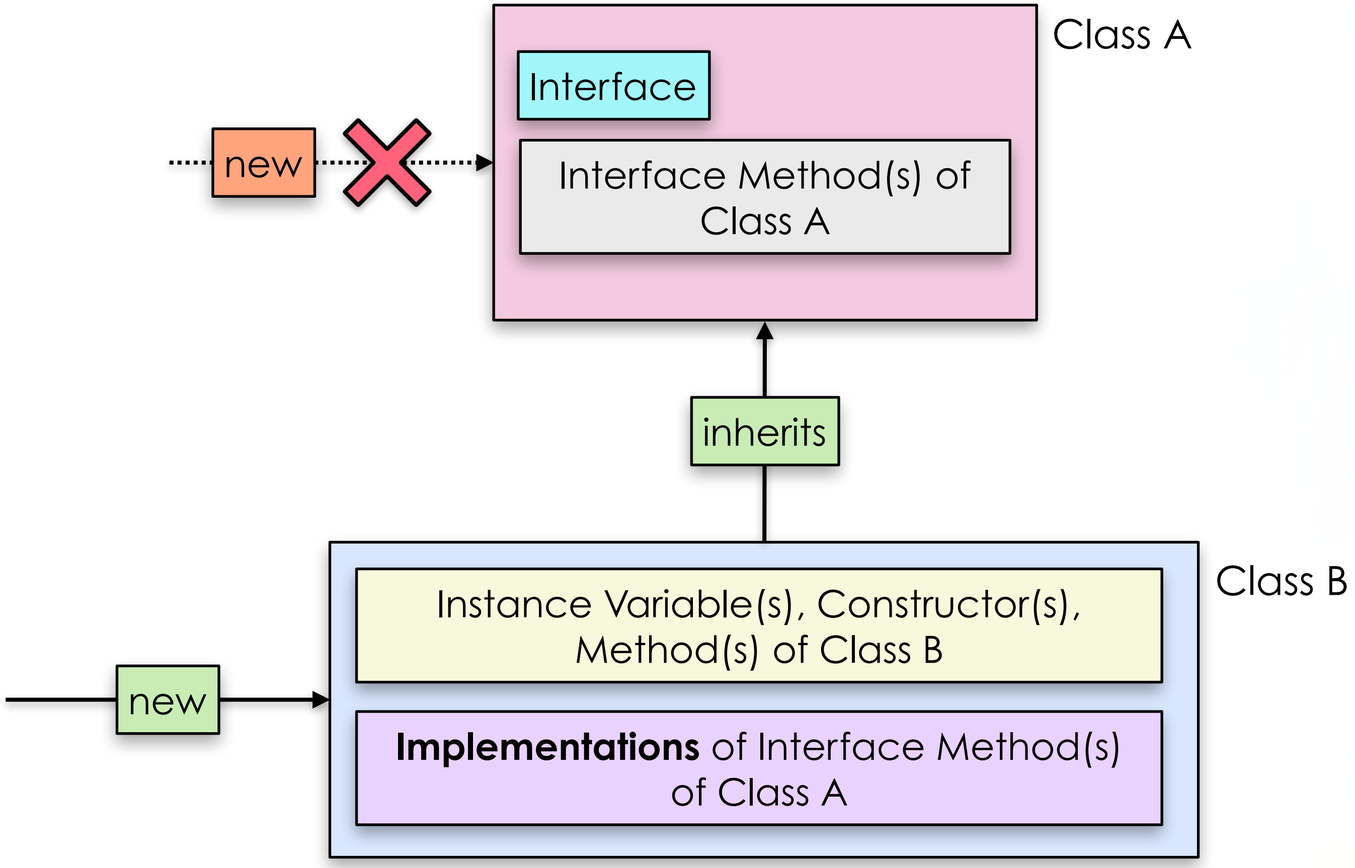
Savings Balance: $150
Checking Balance: $500
Daily Interest: $0

Interface

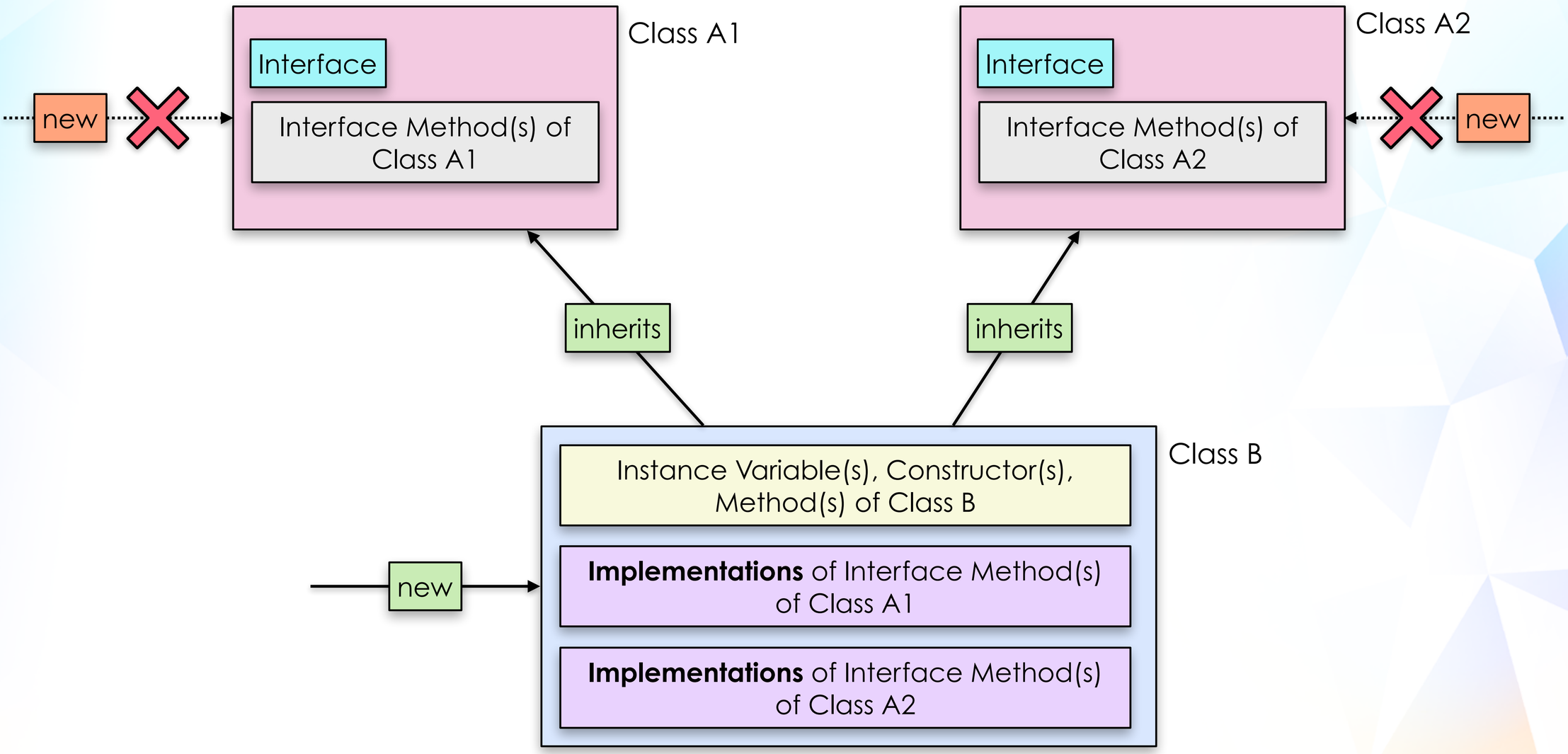
Interface

- An Interface is defined using the **interface** keyword
- An Interface defines only **interface method(s)** (not implemented methods)
- An Interface **cannot be instantiated** and is designed to be inherited
- A child that inherits an interface must implement **all interface methods** defined by the **Interface**
- A child can inherit from **multiple** interfaces

Interface (Single)



Interface (Multiple)



Interface

A sample C# Interface looks like this

Interface Name

```
public interface InterfaceA
{
    public string InterfaceMethodA();
}
```

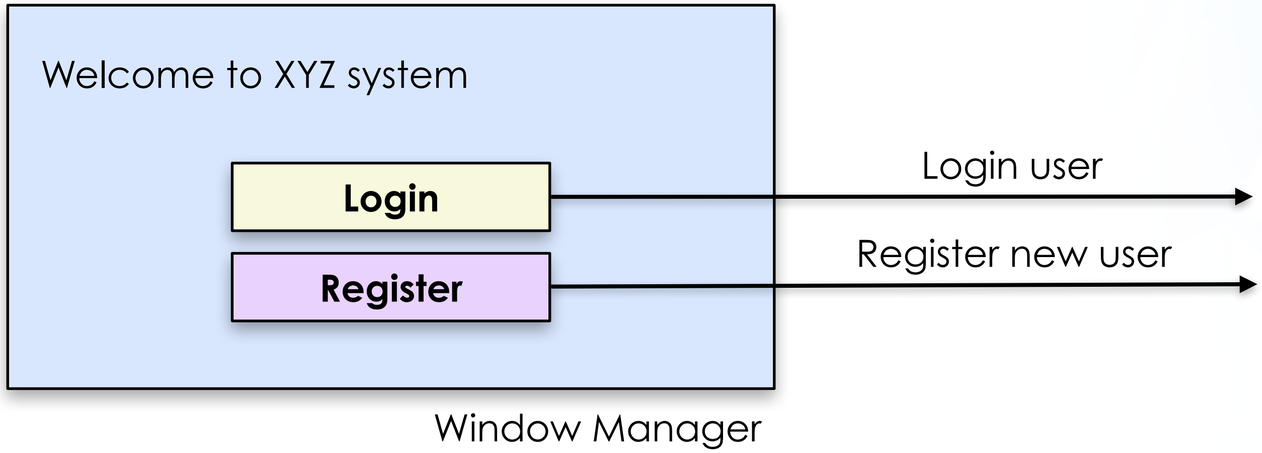
Interface Method

Example: Detect Mouse Clicks

- Consider a Window Manager (WM) that can contain a set of text-labeled, clickable buttons
- The WM is responsible to detect a mouse-click on its buttons and inform the button when it has been clicked
- To simplify and standardize communication, the WM can leverage on an Interface, such that buttons implementing that Interface can be notified when they have been clicked

Example: Detect Mouse Clicks

Let's explore how a Login button and a Register button can receive mouse-clicks from our Window Manager



IClickable Interface

A sample Interface for receiving mouse-click notifications

Uses the keyword **interface**

Name of the interface

```
public interface IClickable
{
    public void OnClick();
}
```

Name of an **interface method**

An **interface** can define **multiple interface methods**

Our Buttons

Implementing the IClickable Interface in our buttons

```
public class Button : IClickable
{
    private string label;

    public Button(string label)
    {
        this.label = label;
    }

    public virtual void onClick()
    {
    }
}
```

inherits

```
public class LoginBtn : Button
{
    public LoginBtn(string label) : base(label)
    {
    }

    public override void onClick()
    {
        // login user
    }
}
```

inherits

```
public class RegisterBtn : Button
{
    public RegisterBtn(string label) : base(label)
    {
    }

    public override void onClick()
    {
        // register new user
    }
}
```

Our Window Manager

The Window Manager detects mouse clicks and dispatch click-notifications to the respective buttons

```
public class WindowManager
{
    public void Start()
    {
        while (true) {
            if (OS.GetMouseState() == OS.ID_MOUSECLICK) {
                Button btn = HitTest();
                if (btn != null) {
                    btn.onClick();
                }
            }
        }
    }
}
```

The OS static class (code not shown) contains mouse information that the Window Manager can query

The HitTest() method (code not shown) returns the current button that was clicked or null if none were clicked

Example: Logging

- Consider a Log class that logs the current states of your objects when your program runs
- The Log class does not need to know the inner workings of your objects to get their data for logging
- An Interface can be defined and acts as a **contract** between the Log class and the objects; objects that need logging simply implements the interface method(s) that the Log class can call, to return the states of the objects for logging

ILoggable Interface

Defining and Implementing a Logging Interface

```
public interface ILoggable
{
    public void GetData();
}
```

Our logging interface

```
public class MyClass : ILoggable {
    private int x;

    public MyClass(int x) {
        this.x = x;
    }

    public void Add(int x) {
        this.x += x;
    }

    public string GetData() {
        return x.ToString();
    }
}
```

Implemented the interface method for InterfaceA

Our class that implemented the logging interface

Interface

Another class that implements the **ILoggable** interface

```
public class MyClass2 : ILoggable {
    private List<int> list = new List<int>();

    public void Add(int x) {
        list.Add(x);
    }

    public string GetData()  {
        string str = "";

        foreach (int x in list) {
            str += x + " ";
        }

        return str;
    }
}
```

Implemented the interface method for InterfaceA

Interface

The Log class using ILoggable interface in its design

```
public class Log {
    private System.IO.StreamWriter writer;

    public Log(string path) {
        writer = File.CreateText(path);
        writer.AutoFlush = true;
    }

   public void Write(ILoggable obj) {
        writer.WriteLine(obj.GetData());
    }
}
```

Relies on InterfaceA

Interface

Leveraging on ILoggable interface to get the object's data (GetData())

```
MyClass obj1 = new MyClass(10);
obj1.Add(5);
obj1.Add(10);

MyClass2 obj2 = new MyClass2();
obj2.Add(5);
obj2.Add(10);

Log log = new Log("/tmp/log.txt");
log.Write(obj1);
log.Write(obj2);
```

Notice that the Log object let the different classes define how they want to format their data

File Content

25
5 10

The End