Encapsulation and Polymorphism

Encapsulation, Polymorphism, Class Hierarchies, Cohesion and Coupling

SoftUni Team

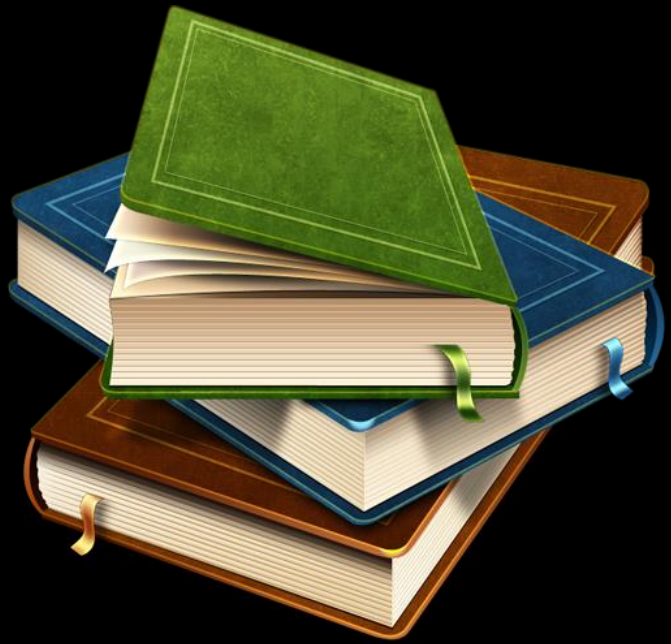
Technical Trainers

Software University

http://softuni.bg

Contents

1. Encapsulation
2. Polymorphism
3. Class Hierarchies: Real World Example
4. Cohesion and Coupling



Encapsulation

Encapsulation

- Encapsulation hides the implementation details
- Class announces only a few operations (methods) available for its clients – its public interface
- All data members (fields) of a class should be hidden
  - Accessed via properties (read-only and read-write)
- No interface members should be hidden
- Encapsulation == hide (encapsulate) data behind constructors and properties

Encapsulation – Example

- Data fields are **private**
- Constructors and accessors are defined (getters and setters)

```
Person
-name : string
-age : int
+Person(string name, int age)
+Name : string { get; set; }
+Age : TimeSpan { get; set; }
```

Encapsulation in C#

SOFTWARE UNIVERSITY FOUNDATION

- Fields are always declared **`private`**
  - Accessed through properties in read-only or read-write mode
  - Properties perform checks to fight invalid data
- Constructors are declared **`public`**
  - Constructors perform checks to keep the object state valid
- Interface methods are always **`public`**
  - Not explicitly declared with **`public`**
- Non-interface methods are declared **`private`** / **`protected`**

Encapsulation – Benefits

- Ensures that structural changes remain local
  - Changing the class internals does not break any outside code
  - Allows changing the internal class implementation
- Encapsulation allows adding logic when accessing object data
  - E.g. validations on when a property is modified
  - E.g. notifications about changes (allows data binding)
- Hiding implementation details reduces complexity
  - Easier maintenance

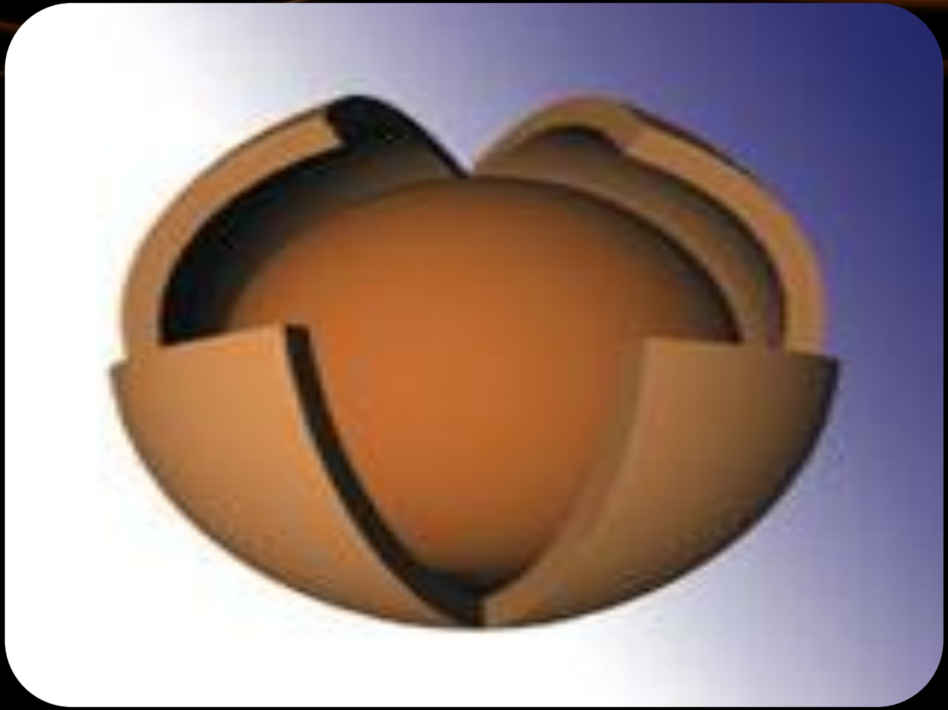
Encapsulation – Example

SOFTWARE UNIVERSITY FOUNDATION

```
public class Person
{
  private string name;

  public string Name
  {
    get { return this.name; }
    set
    {
      if (value == null)
        throw new ArgumentException("Invalid person name.");
      this.name = value;
    }
  }
}
```

The field "name" is encapsulated (hidden)

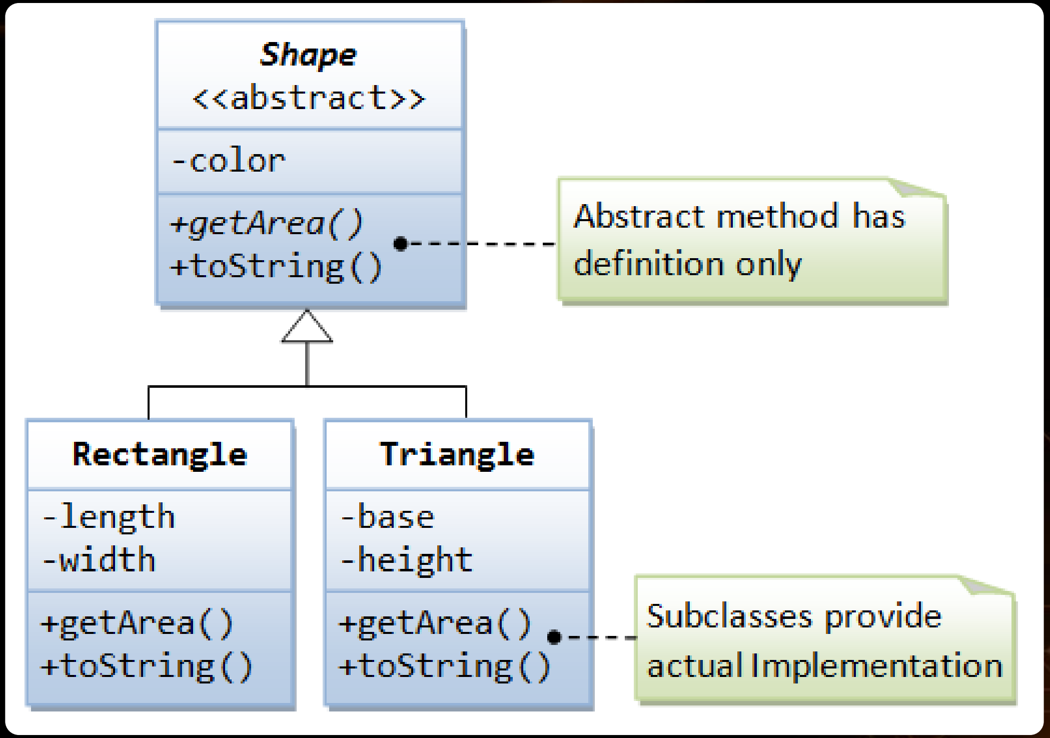


Encapsulation

Live Demo

SOFTWARE UNIVERSITY FOUNDATION

Exercise in Class

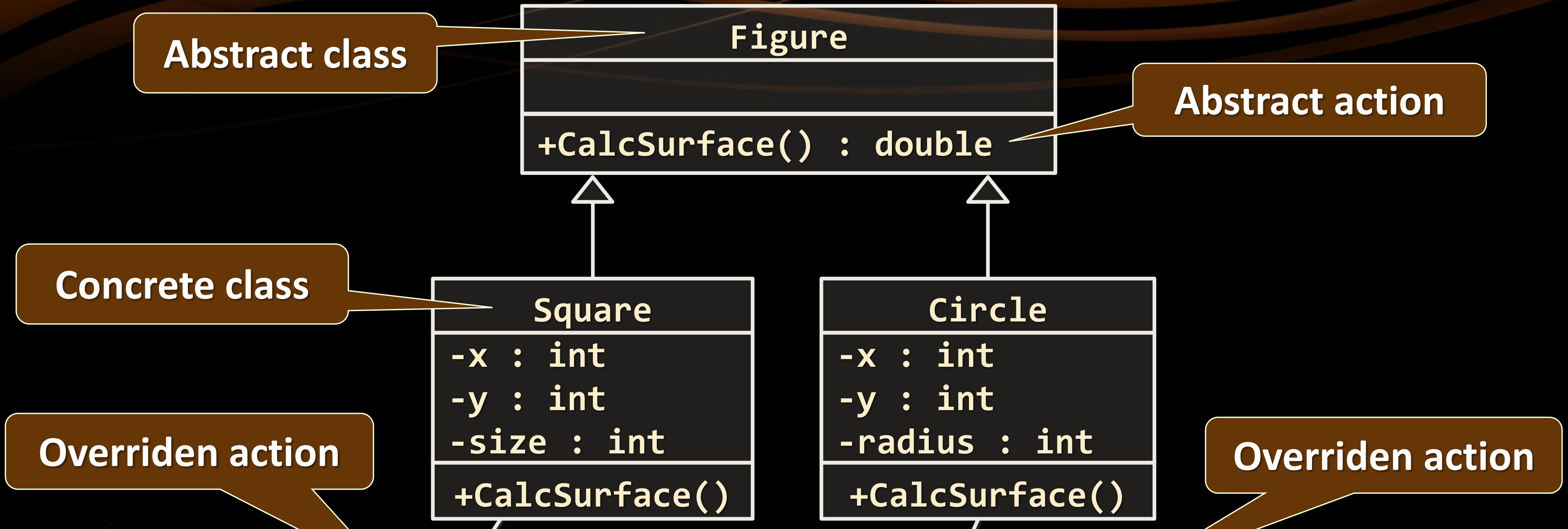


Polymorphism

Polymorphism

- Polymorphism = the ability to take more than one form (objects have more than one type)
  - A class can be used through its parent interface
  - A child class may override (change) some of the parent's methods
- Polymorphism allows invoking abstract operations
  - Defined in the base class / interface
  - Implemented in the child classes
  - Declared as **`abstract`** or **`virtual`** or inside an interface

Polymorphism – Example




```
public override double CalcSurface()
{
    return size * size;
}
```

```
public override double CalcSurface()
{
    return Math.PI * radius * raduis;
}
```

Polymorphism – Example (2)

SOFTWARE UNIVERSITY FOUNDATION

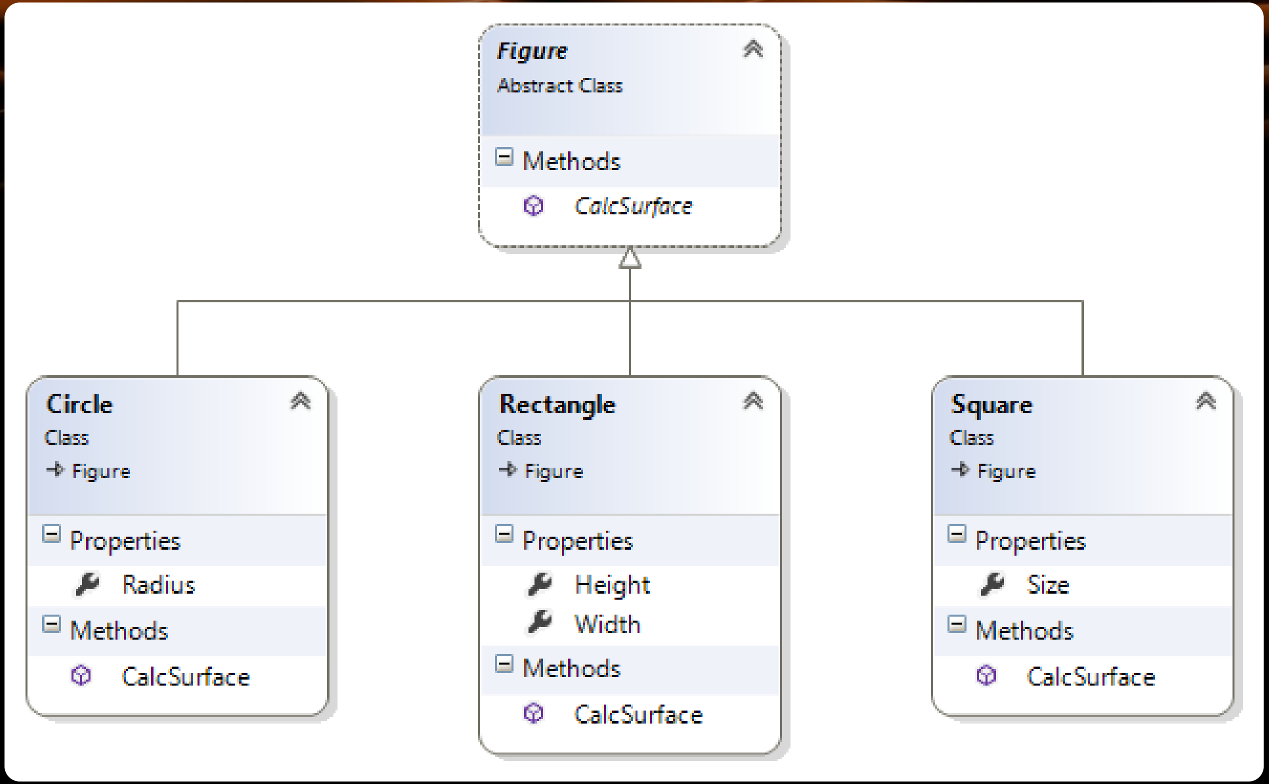
```
abstract class Figure
{
    public abstract double CalcSurface();
}

class Square
{ public override double CalcSurface() { return size * size; } }

class Circle
{ public override double CalcSurface() { return PI * r * r; } }

Figure f1 = new Square(…);
Figure f2 = new Circle(…);

double surface = f1.CalcSurface(); // Call Square.CalcSurface()
double surface = f2.CalcSurface(); // Call Circle.CalcSurface()
```



Polymorphism

Live Demo

Polymorphism – Why?

- Why handle an object of given type as object of its base type?
  - To invoke abstract operations implemented in the child classes
  - To mix different data types in the same collection
    - E.g. `List<Figure>` can hold `Circle` and `Rectangle` objects
  - To pass more specific object to a method that expects a more generic type (e.g. `SchoolStudent` instead of `Student`)
  - To declare a more generic field which will be initialized and "specialized" later (in a subclass)

Virtual Methods

- A virtual method is:
  - Defined in a base class and can be changed (overridden) in the descendant classes
- Virtual methods are declared through the keyword **virtual**

```
public virtual void Draw() { … }
```

- Methods declared as virtual in a base class can be overridden using the keyword **override**

```
public override void Draw() { … }
```

Virtual Methods – Example

```
abstract class Figure
{
    public virtual void Draw()
    {
       Console.WriteLine(
         "I am a figure of type: {0}", this.GetType().Name);
    }
}

class Circle : Figure
{
    public override void Draw()
    {
        Console.WriteLine("I am a circle");
    }
}
```

Calling Base Virtual Methods – Example

SOFTWARE UNIVERSITY FOUNDATION

```
class Circle : Figure
{
    public override void Draw()
    {
        Console.WriteLine("I am a circle:");
        Console.WriteLine(" --- ");
        Console.WriteLine("|   |");
        Console.WriteLine(" --- ");
    }
}

class SpecialCircle : Circle
{
    public override void Draw()
    {
        Console.WriteLine("I am a special circle.");
        base.Draw();
    }
}
```

Virtual Methods

Live Demo

More about Virtual Methods

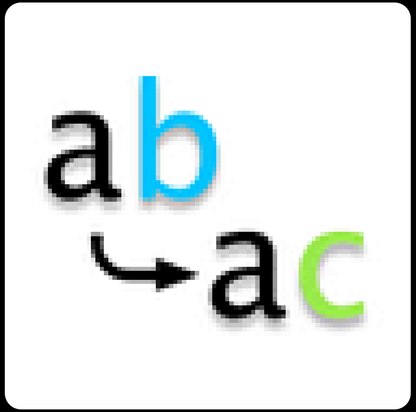
- Abstract methods are purely virtual
  - If a method is **abstract** → it is **virtual** as well
  - Abstract methods are designed to be changed (overridden) later
- Interface members are also purely virtual (abstract)
  - They have no default implementation and are designed to be overridden in descendant classes
- Virtual methods can be hidden through the **new** keyword:

```
public new double CalculateSurface() { return … }
```

The override Modifier

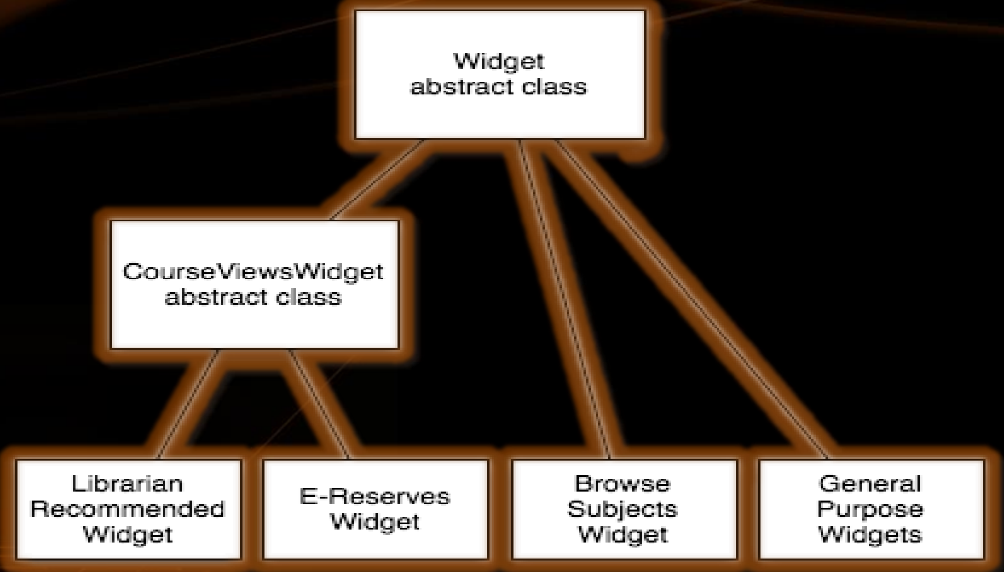
SOFTWARE UNIVERSITY FOUNDATION

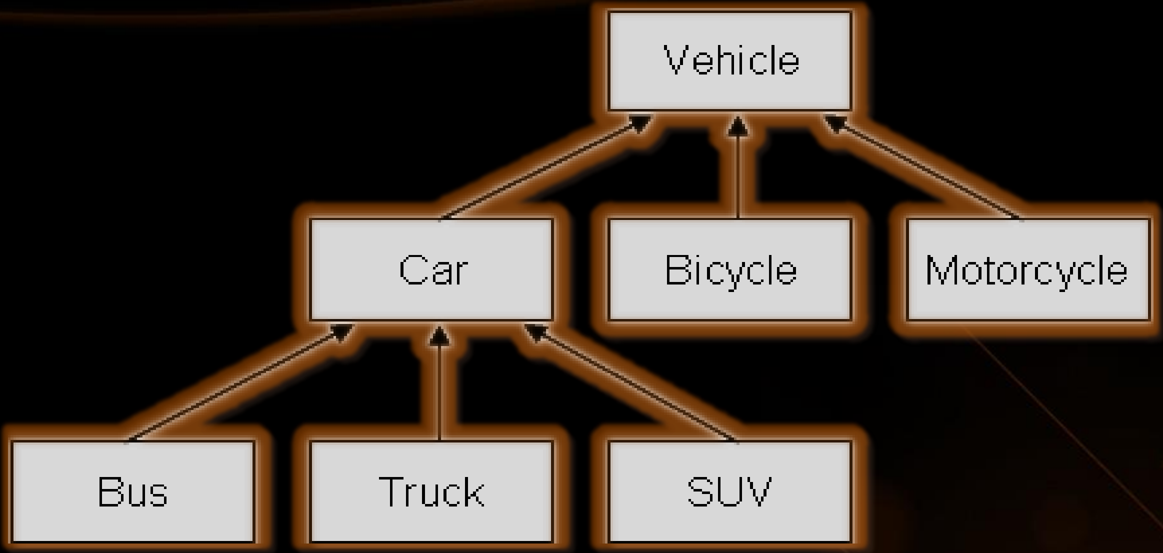
- Use **override** to modify a method or property
  - Provide a replacement implementation for the inherited member
  - You cannot override a non-virtual or static method
- The overridden base method must be one of the following:
  - **virtual**
  - **abstract**
  - **override**
  - (interface method)



Polymorphism – How It Works?

- Polymorphism ensures that the appropriate method of the subclass is called through its base class' interface
- In C++, C#, Java polymorphism is implemented using a technique called "late binding"
- The exact method to be called is determined at runtime
  - Just before performing the call
  - Applied for all **`abstract`** / **`virtual`** methods
- Note: late binding is a bit slower than normal (early) binding

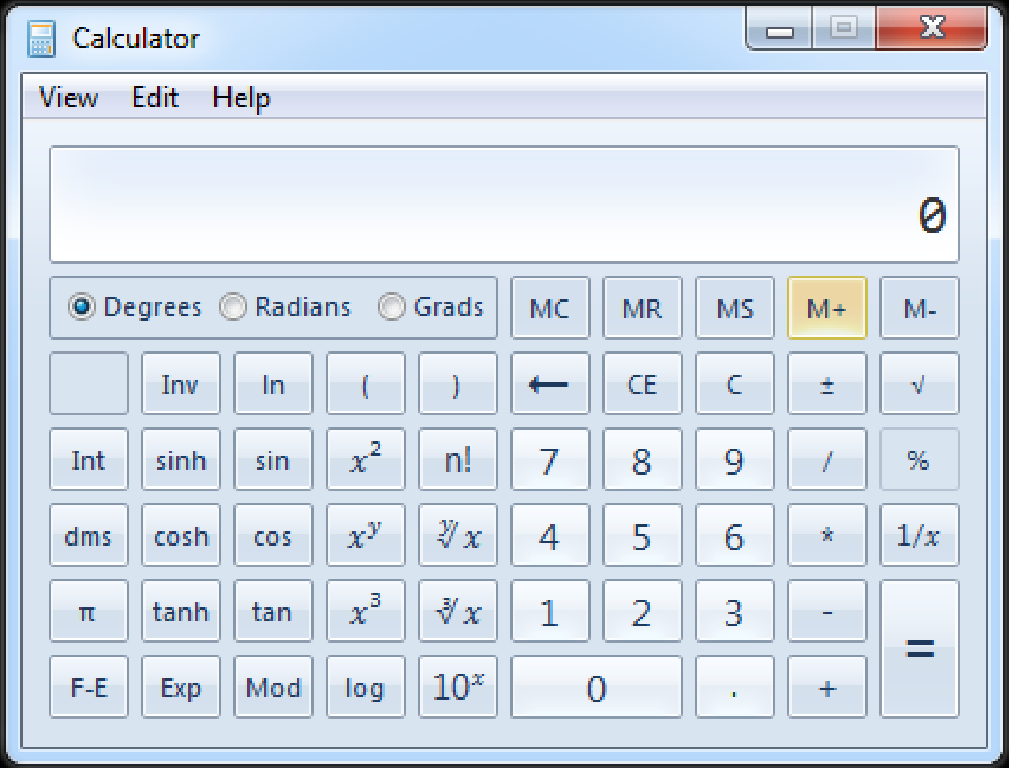




Class Hierarchies: Real World Examples

Real World Example: Calculator

- Creating an application like the Windows Calculator
  - Typical scenario for applying the object-oriented approach



Real World Example: Calculator (2)

- The calculator consists of controls:
  - Buttons, text boxes, menus, check boxes, panels, etc.
- Class `Control` – the root of our OO hierarchy
  - All controls can be painted on the screen
    - Should implement an interface `IPaintable` with a method `Paint(surface)`
  - Common control properties:
    - Location, size, text, face color, font, background color, etc.

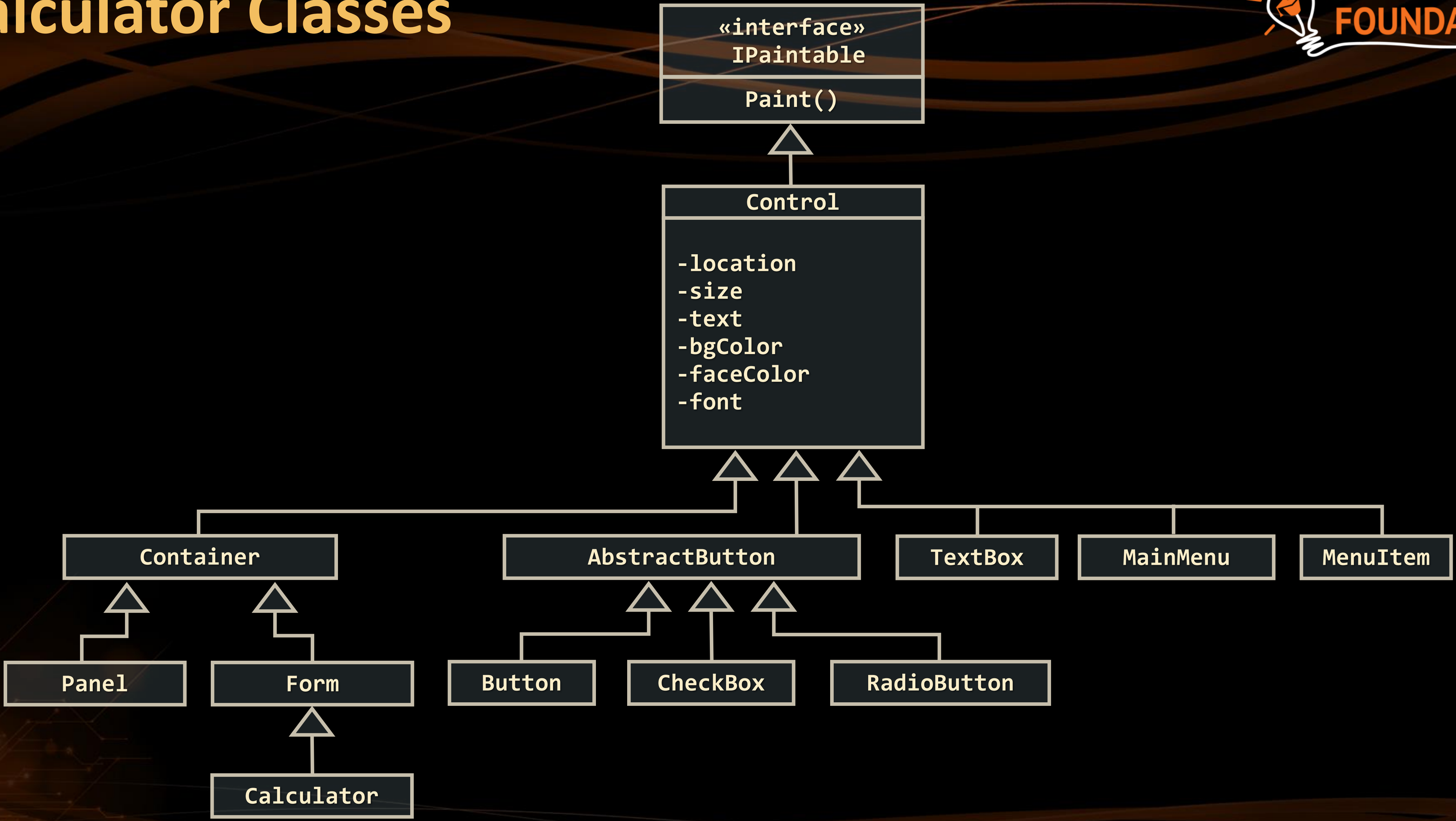
Real World Example: Calculator (3)

- Some controls could contain other (nested) controls inside
  - E.g. panels and toolbars can hold other controls
  - Class **`Container`** – extends **`Control`**, holds a list of child controls
- The **`Calculator`** itself is a **`Form`**
  - **`Form`** is a special kind of **`Container`**
  - Forms hold also border, title, icon and system buttons
  - The form title is the **`text`** derived from **`Control`**
- How does **`Calculator`** paint itself?
  - Invokes **`Paint()`** for all child controls inside it

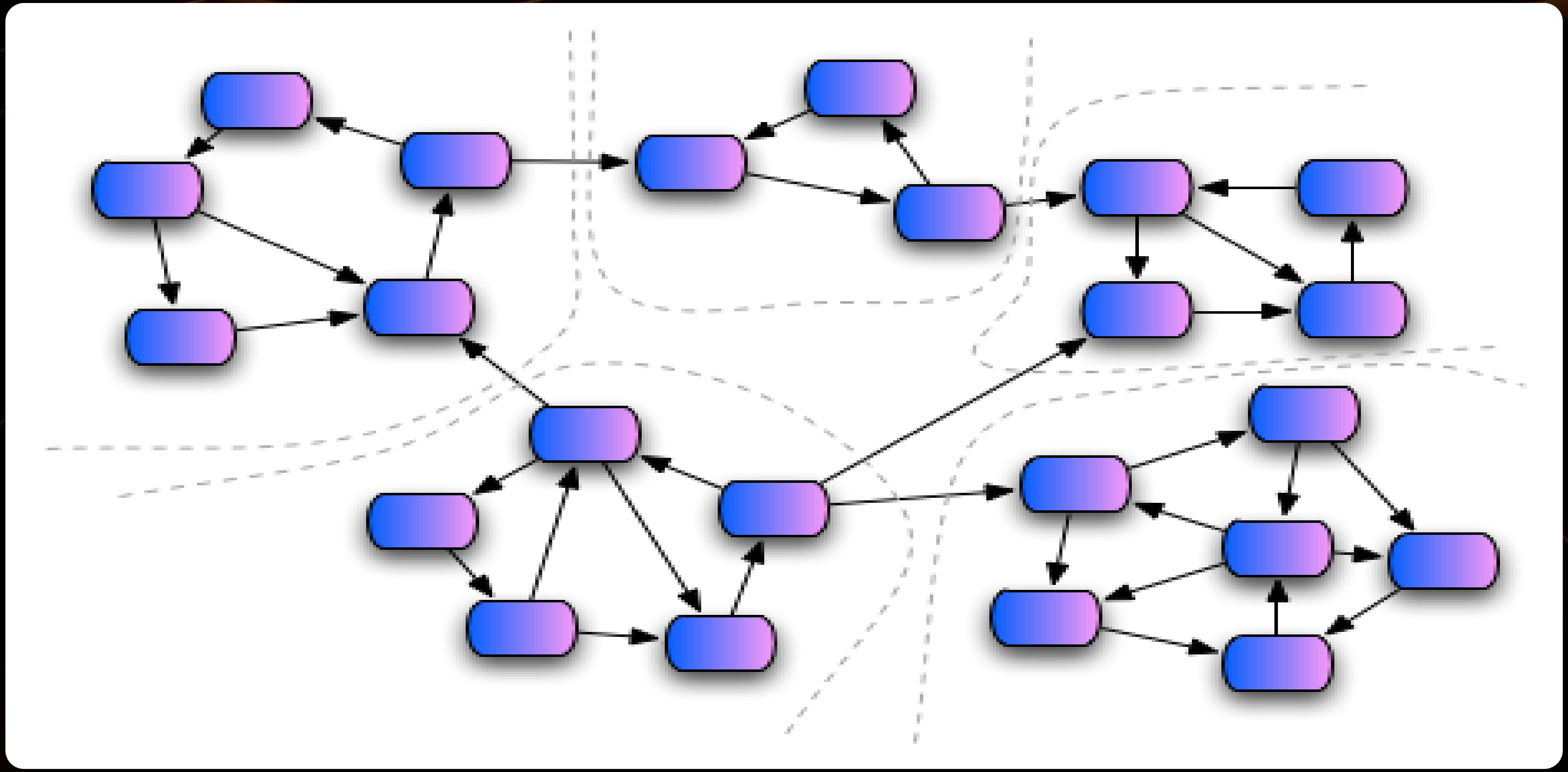
Real World Example: Calculator (4)

- How does a **`Container`** paint itself?
  - Invokes **`Paint()`** for all controls inside it (chain of responsibility)
  - Each control knows how to visualize (paint) itself
- Buttons, check boxes and radio buttons are similar
  - Can be pressed
  - Can be focused
- All buttons could derive from a common parent class **`AbstractButton`**

Calculator Classes



Exercise in Class



Cohesion and Coupling

Cohesion

- Cohesion describes
  - How closely the routines in a class or the code in a routine support a central purpose
- Cohesion must be strong
  - Well-defined abstractions keep cohesion strong
- Classes must contain strongly related functionality and aim for single purpose
- Cohesion is a powerful tool for managing complexity

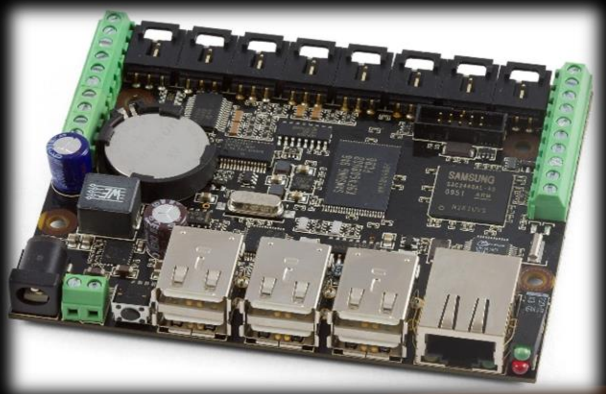
Good and Bad Cohesion

- Good cohesion: HDD, CR-ROM, remote control



- Bad cohesion: spaghetti code, single-board computer





Strong Cohesion

- Strong cohesion (good cohesion) example:
  - Class **`Math`** that has methods:

    `Sin()`, **`Cos()`**, **`Asin()`**, **`Sqrt()`**, **`Pow()`**, **`Exp()`**, **`Math.PI`**, **`Math.E`**

```
double sideA = 40, sideB = 69;
double angleAB = Math.PI / 3;

double sideC = sideA * sideA + sideB * sideB -
    2 * sideA * sideB * Math.Cos(angleAB);

double sidesSqrtSum =
    Math.Sqrt(sideA) + Math.Sqrt(sideB) + Math.Sqrt(sideC);
```

Weak Cohesion

- Weak cohesion (bad cohesion) example
  - Class **Magic** that has these methods:

```
public void PrintDocument(Document d);
public void SendEmail(
  string recipient, string subject, string text);
public void CalculateDistanceBetweenPoints(
  int x1, int y1, int x2, int y2)
```

- Another example:

```
MagicClass.MakePizza("Fat Pepperoni");
MagicClass.WithdrawMoney("999e6");
MagicClass.OpenDBConnection();
```

Coupling

- Coupling describes how tightly a class or a routine is related to other classes or routines
- Coupling must be kept loose
  - Modules must depend little on each other
    - Or be entirely independent (loosely coupled)
  - All classes / routines must have small, direct, visible, and flexible relationships to other classes / routines
  - One module must be easily used by other modules

Loose and Tight Coupling

- Loose coupling:
  - Easily replace old HDD
  - Easily place this HDD to another motherboard

- Tight coupling:
  - Where is the video card?
  - Can you change the audio controller?

Loose Coupling – Example

```
class Report : IReport
{
    public bool LoadFromFile(string fileName) {…}

    public bool SaveToFile(string fileName) {…}
}

class Printer
{
    public static int Print(IReport report) {…}
}

class Program
{
    static void Main()
    {
        Report myReport = new Report();
        myReport.LoadFromFile(@"C:\Reports\DailyReport.xml");
        Printer.Print(myReport);
    }
}
```

Tight Coupling – Example

SOFTWARE UNIVERSITY FOUNDATION

```
class MathParams
{
    public static double operand;
    public static double result;
}
class MathUtil
{
    public static void Sqrt()
    {
      MathParams.result = CalcSqrt(MathParams.operand);
    }
}
class MainClass
{
    static void Main()
    {
        MathParams.operand = 64;
        MathUtil.Sqrt();
        Console.WriteLine(MathParams.result);
    }
}
```

Spaghetti Code

- Combination of bad cohesion and tight coupling:

```
class Report
{
    public void Print() {…}
    public void InitPrinter() {…}
    public void LoadPrinterDriver(string fileName) {…}
    public bool SaveReport(string fileName) {…}
    public void SetPrinter(string printer) {…}
}
class Printer
{
    public void SetFileName() {…}
    public static bool LoadReport() {…}
    public static bool CheckReport() {…}
}
```

Exercise in Class

Summary

- Encapsulation hides internal data
  - Access through constructors and properties
  - Keeps the object state valid
- Polymorphism == using objects through their parent interface
  - Allows invoking abstract actions overridden in a child class
- Strong cohesion == single purpose
- Loose coupling == minimal interaction with others

OOP – Encapsulation and Polymorphism

SOFTWARE UNIVERSITY FOUNDATION

Questions?

https://softuni.bg/courses/oop/

License

Free Trainings @ Software University

- Software University Foundation – softuni.org
- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University @ YouTube
  - youtube.com/SoftwareUniversity
- Software University Forums – forum.softuni.bg