

Professional Software Engineering

Andrea Carrara and Patrick Berggold

Hritik Singh and Mohab Hassaan – Tutors

Chair of Computational Modeling and Simulation

Schedule Week 2

- » Functions & References
- » Object Oriented Programming
 - Inheritance
 - Abstract Classes



RECAP LAST LECTURE

Datatypes

» Value types

- bool (true or false)
- char (16-bit Unicode)
- Numeric Types
- enum
- struct

» Reference types

- object
- strings
- array
- and everything derived (classes)

» Strings are immutable !

if & switch statements

```
» if( condition )
» {
»     statement block...
» }
» else
» {
»     statement block...
» }
```

```
switch(var) {
    case one:
        // do something
        break;
    case two:
    case three:
        // do something if one is true
        break;
    default:
        // is executed if there is no match
        // no break needed
}
```

while & for loops

```
» while (condition)  
  
» {  
  
»     statement block...  
  
» }
```

```
» for (initializer; condition; iterator)  
  
» {  
  
»     statement block...  
  
» }
```

Both *while* and *for* loop as long as condition is fulfilled.

Loops: break & continue keywords

```
» while (condition)
» {
»     statement block...
» }
```

- » **break**: used to jump out of a loop
- » **continue**: continue with the next iteration in a loop

```
int counter = 0;
int limit = 5;
while (true) {
    if (counter > limit){
        // step out of loop
        break;
    }
    counter++;
}
```

[Live coding](#)

7

Casting - Example

```
public void CastingExample() {  
    double floatingPoint_1 = 1.4;  
    double floatingPoint_2 = 1.8;  
    int result = 0;  
  
    result = (int) floatingPoint_1;  
    //result will now be 1  
    result = (int) floatingPoint_2;  
    //result will still be 1  
    floatingPoint_2 = (double) result;  
    // floatingPoint_2 is 1  
}
```




REFERENCES & FUNCTIONS

Basics: Functions | Procedures

- » What are functions? Basically, a code package... → functions are ideal for repetitive work!

Example:

```
datatype Name(ParameterA, ParameterB, ...) {  
    //Code  
    return returnValue  
}
```

- » Function have multiple *inputs*, but one *output*
- » *datatype* specifies the return type

Basics: Functions | Procedures

- » What are functions? Basically, a code package... → functions are ideal for repetitive work!

Example:

```
datatype Name(ParameterA, ParameterB, ...) {  
    //Code  
    return returnValue  
}
```

- » Function have multiple **inputs**, but one **output**
- » *datatype* specifies the return type

Input and output variables have a datatype (like *int*, *string*, *float*);
If no value is returned from the function the keyword **void** is used!


- » A function is identified by its signature
 - There can be multiple functions of the same Name
 - This is called overloading a function

Basics: Functions | Why to use functions?

- » Reusability
 - old projects
 - invoke multiple times
 - building blocks
- » Abstraction
 - you don't need to know the internals!
 - can you build a car yourself?



Basics: Functions | Scope of Variables

- » The scope of a variable practically means its visibility
- » Variables defined inside a function or a loop are not visible outside this function (in C#...)

Scope depends on language...
- » Information can only leave the **scope** of the function if:
 - an outside variable is set
 - a value is returned from a function
 - a reference is passed as a parameter

```
int wtf = 42;
```

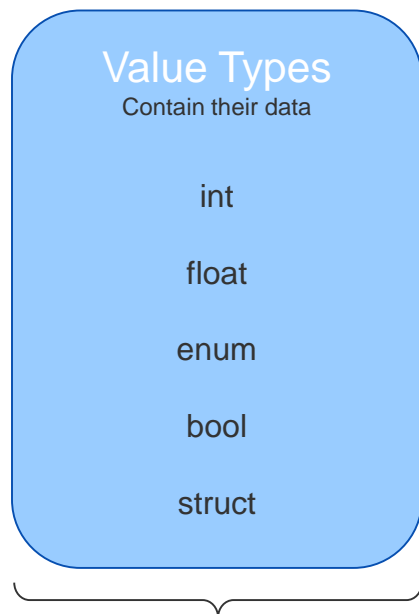
NOT a good name.

Basics: Reference Types

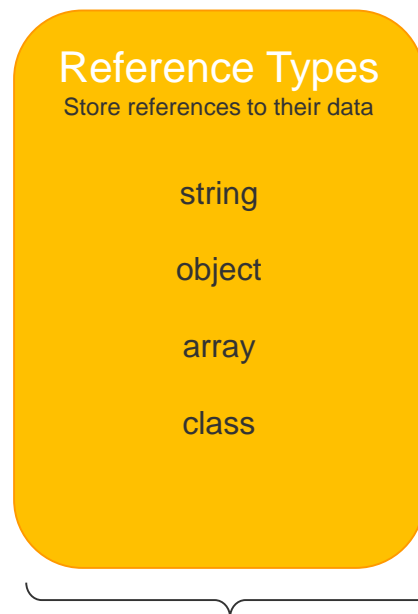
» What is a reference type?

A reference type variable contains a reference to data stored in special part of the memory (aka heap)

» Difference to value types:



Each variable makes a copy of the data



Multiple references to same object possible

Live coding

Basics: Passing by Reference Types

- » What is a reference type?

A reference type variable contains a reference to data stored in special part of the memory (aka heap)

- » Difference to value types:

Value types: contain their data

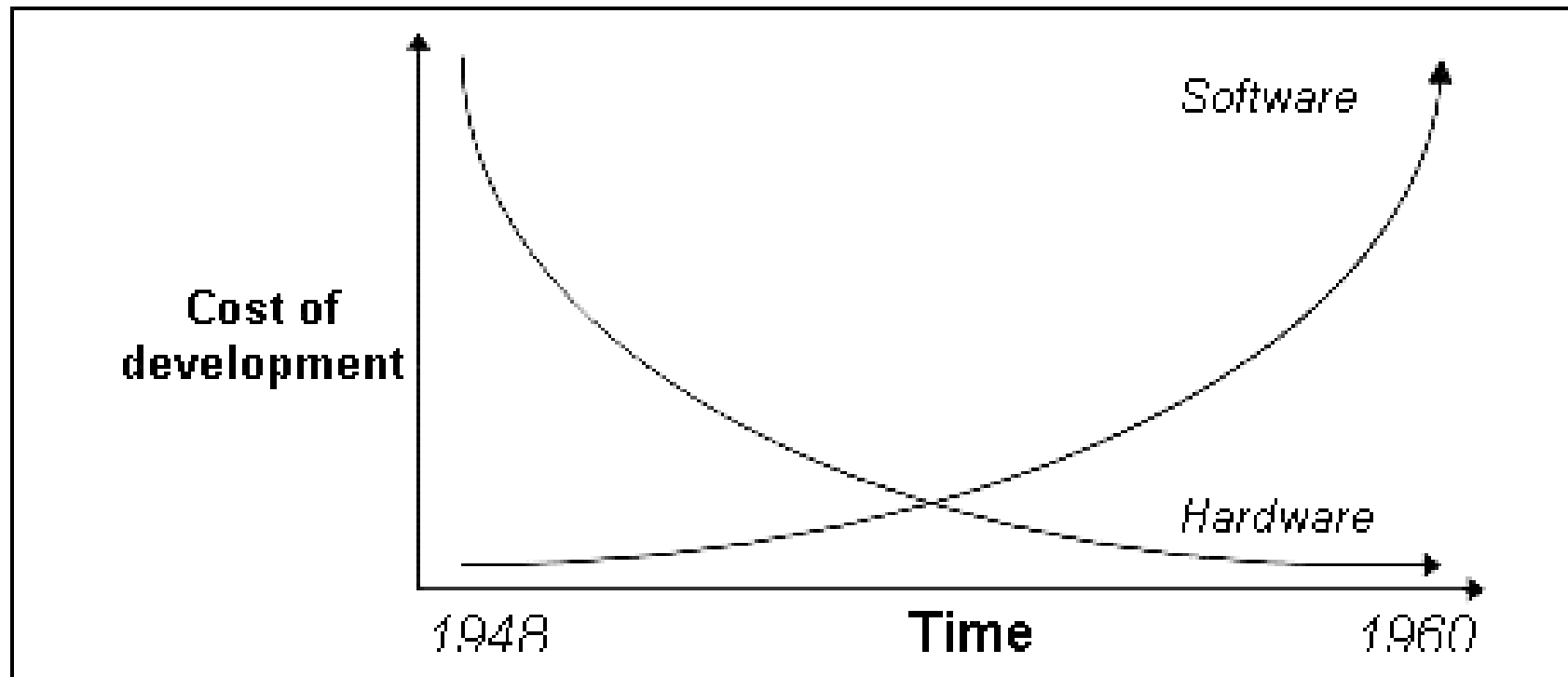
Reference types: Store references to their data

- » Strings are immutable, and behave mostly like value types
- » Passing arguments as reference types via the **ref** or **out** keywords (**ref** requires parameter to be already initialized, **out** does not) will surpass the local scope of variables

An Introduction

OBJECT-ORIENTED PROGRAMMING (OOP)

Motivation: Software Crisis



With cheaper and faster hardware, the code base grew tremendously...

“[The major cause of the software crisis is] that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.”

Edsger W. Dijkstra: The humble programmer

Solution - OOP

- » To handle complex projects, code is broken down to comprehensible and isolated fragments (= objects)
- » The objects are instances of categories (=classes), e.g. 'User', 'Car', 'Circle' or 'UserController'
- » OOP improves Maintainability, Reusability and Comprehensiveness of Code

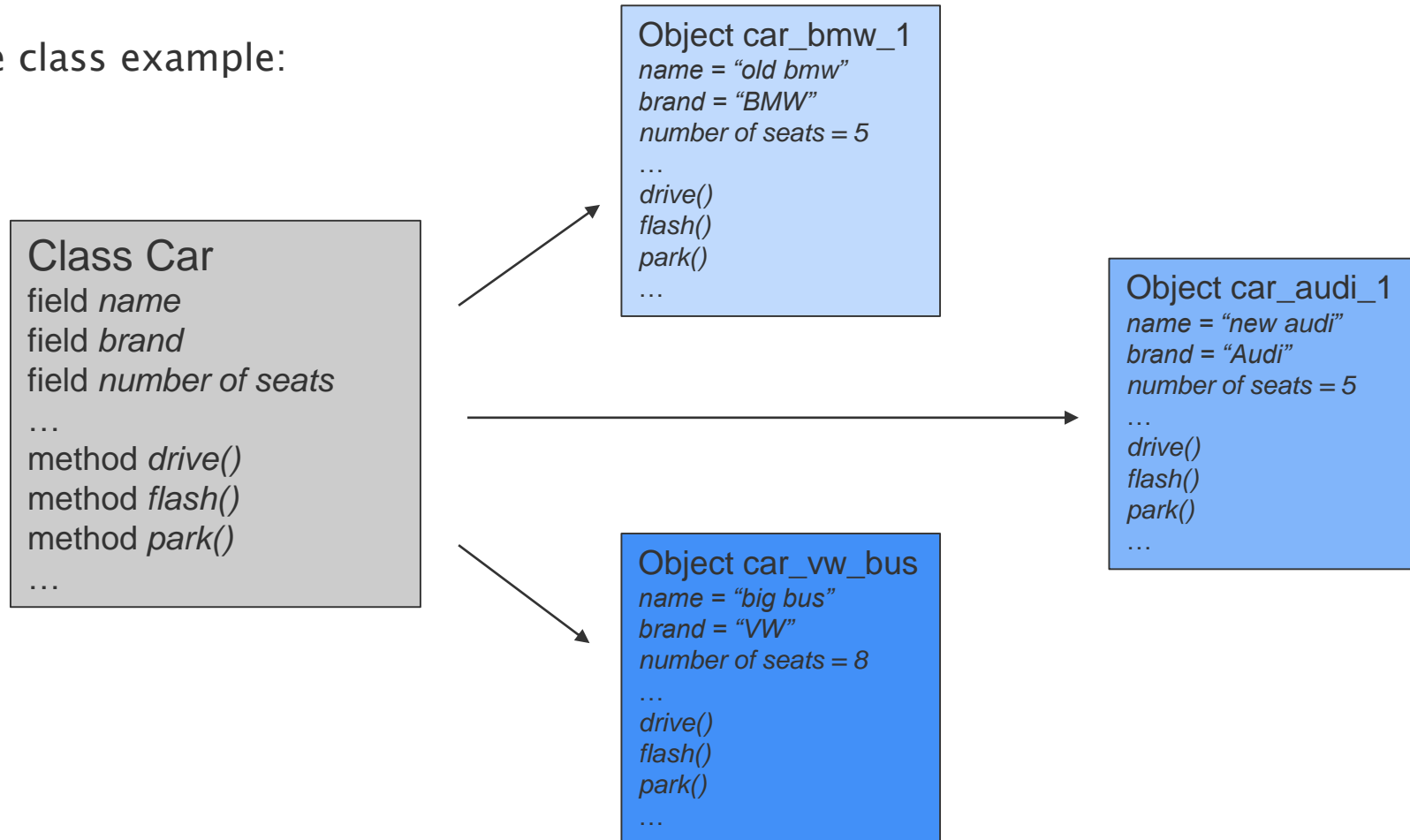


Key Concept And Naming

- » two basic constructs:
 - class
 - object
- » the class is a **blueprint**, or **template**
 - defines the behavior
 - complex **datatype**
- » an object is a “concrete” entity
 - based on defined class
 - independent from other objects
- » function bound to objects are often called **methods**
- » in C# attributes only visible to the object itself are called **fields**
- » Fields and methods are oftentimes referred to as class **members**

Key Concept And Naming

» A simple class example:

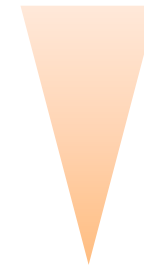


Encapsulation

- » Data hiding is important
- » The programmer does not need to know about internals
- » Communication via available interface (public)

- » Access modifiers
 - `public` → members can be seen on the outside
 - `protected` → members can be seen to the object and its derived classes
 - `private` → members elements can only be seen to the object itself

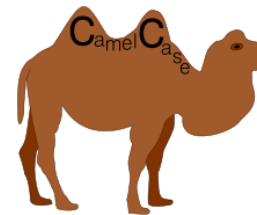
accessibility



Class Declaration in C#

```
class Vector2D {  
    // fields = member variables  
    public double x;  
    public double y;  
  
    // methods = member functions  
    public double Norm() {  
        double nrm = Math.Sqrt(x * x + y * y);  
        return nrm;  
    }  
}
```

1. Use UpperCamelCase for class names
2. use lowerCamelCase for fields
3. Use UpperCamelCase for methods



**Please pay
attention to
this in your
Assignment!**

Object Instantiation and Use

```
Vector2D vec1 = new Vector2D();  
vec1.x = 5;  
vec2.y = 10;  
Console.WriteLine(vec1.Norm());
```

Create Object

Assign public field

Access public method

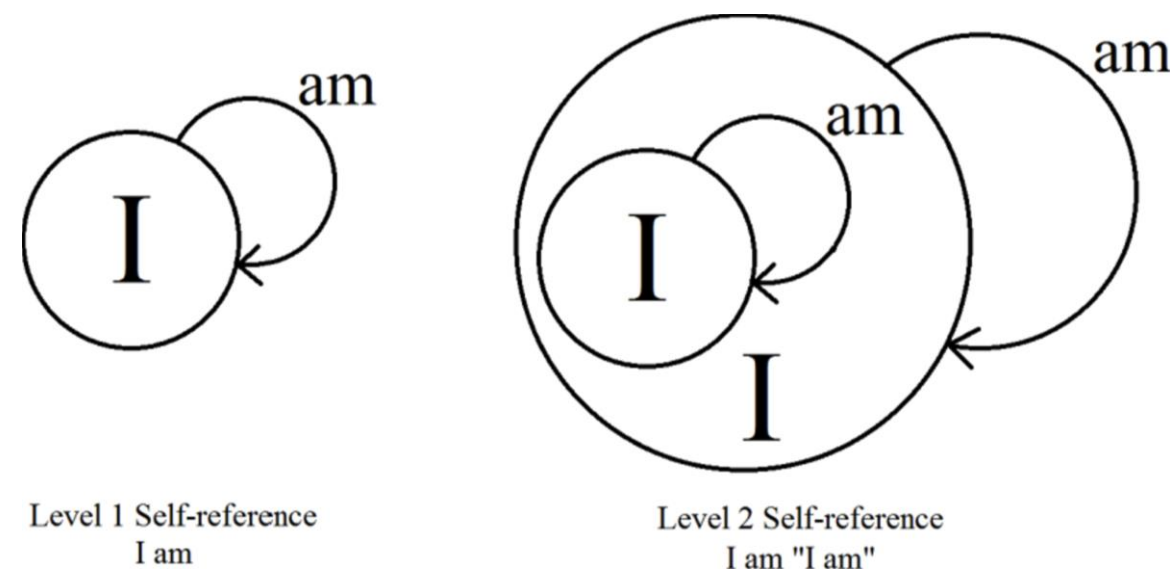
Constructor

- » Special method(s) that is always called automatically when the object is created
- » Method name = class name
- » If no constructor is defined, the compiler automatically creates the standard constructor

```
class Circle {  
    private double r;  
    private double x;  
    private double y;  
  
    //hide the standard constructor  
    private Circle() {  
    }  
  
    public Circle(double radius) {  
        this.r = radius;  
    }  
}
```

The Reference *this*

- » Refers to current instance (object)
- » Helps to distinguish local variable and field
- » Can be passed as parameter to a method
- » **this** implicitly passed to every method
 - Not available in static function



Getters & Setters

- » Makes private fields available to the public
- » You can define which fields you want to make accessible!
- » Apply changes with simple function call
- » Important remarks:
 - Naming!
 - Default access modifier set to `private` within classes (so make them `public` explicitly)

```
private string myProperty;  
  
public string GetMyProperty() {  
    return this.myProperty;  
}  
  
public void SetMyProperty(string value) {  
    this.myProperty = value;  
}
```

Properties – A C# Special

- » A property is a field with a get/set block
- » Accessed from outside like a normal field, implemented as a method
- » Advantage: complete control!

```
public void Main() {  
    Circle myCircle = new Circle();  
    // set accessor is called  
    myCircle.Radius = 23;  
    // get accessor is called  
    double x = myCircle.Radius;  
}
```



```
public class Circle {  
    // hidden "backing" field  
    private double radius;  
    // public property  
    public double Radius {  
        get {  
            return this.radius;  
        }  
        set {  
            if(value >= 0){  
                this.radius = value;  
            }  
            else{  
                Console.WriteLine("Illegal value.");  
            }  
        }  
    }  
}
```

Automatic Properties

```
public class Circle {  
    // property  
    public double Radius { get; set; }  
}  
  
public void Main() {  
    Circle myCircle = new Circle();  
  
    // set accessor is called  
    myCircle.Radius = 23;  
  
    // get accessor is called  
    double x = myCircle.Radius;  
}
```

- » automatic properties
- » get and set assessors are automatically generated
- » Backup field is implicitly created

Static Keyword

- » Static method: not associated with an object, unlike regular methods
- » Static class: class which can only contain static members
- » No object of static class!
- » Object related function cannot be static!
 - Constructor
 - Types

```
class SomeClass {  
    private double member = 3;  
  
    public double NormalMethod() {  
        return this.member;  
    }  
  
    public static double StaticMethod() {  
        // return this.x does not work  
        return 3;  
    }  
}  
  
SomeClass instance = new SomeClass();  
instance.NormalMethod(); //Fine  
instance.StaticMethod(); //Won't compile  
  
SomeClass.NormalMethod(); //Won't compile  
SomeClass.StaticMethod(); //Fine
```

Pass an object to a function

```
public void InstallNewCamera(Camera newCamera){  
    this.AddElement((Element) newCamera);  
}
```

What's to see here?

- » Function is available to the outside and has no return type (void)
- » An object of type `Camera` is passed in
- » `AddElement(Element param)` is another member (`this`-keyword) in this class which takes in a parameter of type `Element` (thus casting)

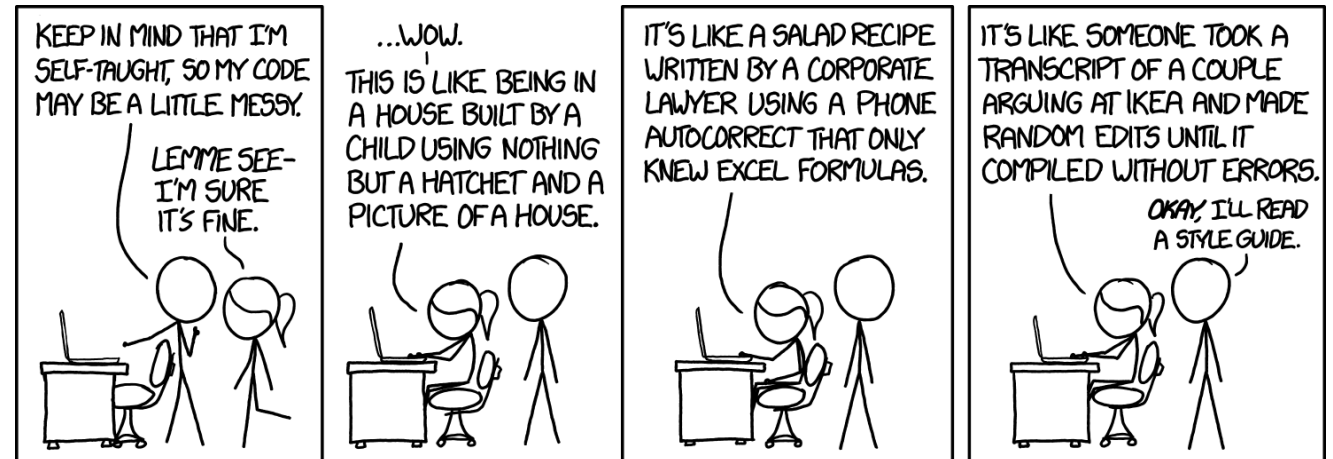
Destructor

- » Same as constructor, but always called if object's memory is freed up (program exits or scope of the object ends)
- » "There can be only one" (*Highlander, 1986*)
- » Tilde operator (~)
- » Cannot be called directly
- » Cannot be inherited

```
class Circle {  
    private double r;  
    private double x;  
    private double y;  
  
    private ~Circle() {  
        // some final code?  
    }  
}
```


Good Programming Practice

- » Maximize adherence
 - One class per file
 - Now what belongs together will grow together (Willi Brandt)
- » Consistency
- » Names should be self-explanatory,



Always choose readability!

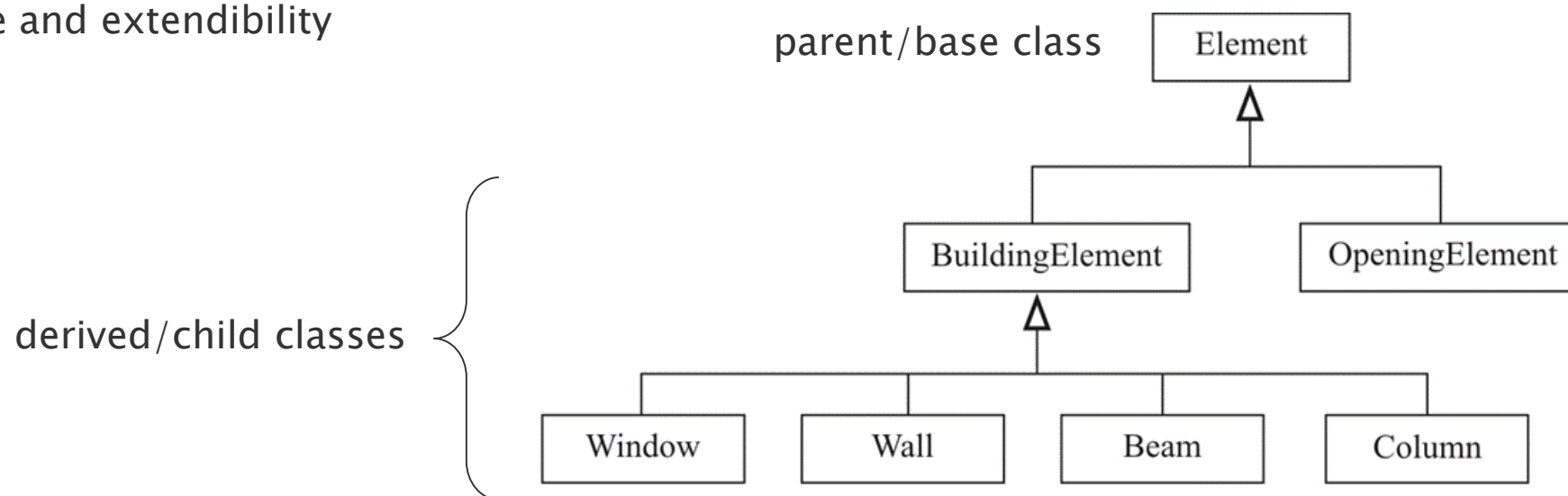


Inheritance, Abstract Classes, Interfaces

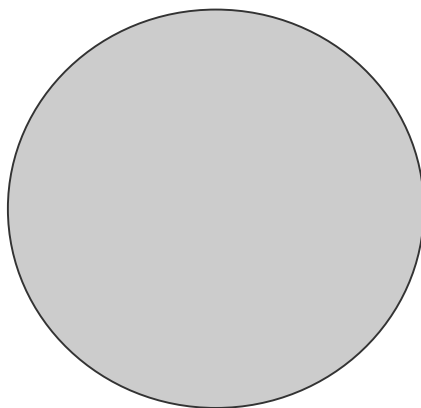
HIERARCHIES IN OOP

Inheritance

- » Powerful concept of object-oriented programming
- » Inheritance hierarchy reflects generalization / refinement
- » Derived classes inherit fields and methods from super-classes
- » Facilitates re-use and extendibility



Example: Figure



» Circle

- 1 sides
- Perimeter($2\pi r$) and Area(πr^2)



» Square

- 4 sides
- Each side has same length
- Perimeter($4x$) and Area(x^2)

Declaration And Use

- » Circle and Rectangle are defined as subclasses of Figure
- » They inherit the properties *PositionX* and *PositionY*
- » They inherit the method Move()
- » When instantiated, base constructor is called first, then derived constructor

```
public class Figure {  
    protected int PositionX { get; set; }  
    protected int PositionY { get; set; }  
  
    public void Move(int dx, int dy) {  
        PositionX = PositionX + dx;  
        PositionY = PositionY + dy;  
    }  
}  
  
public class Circle : Figure {  
    public int Radius { get; set; }  
}  
  
public class Rectangle : Figure {  
    public int A { get; set; }  
    public int B { get; set; }  
}
```

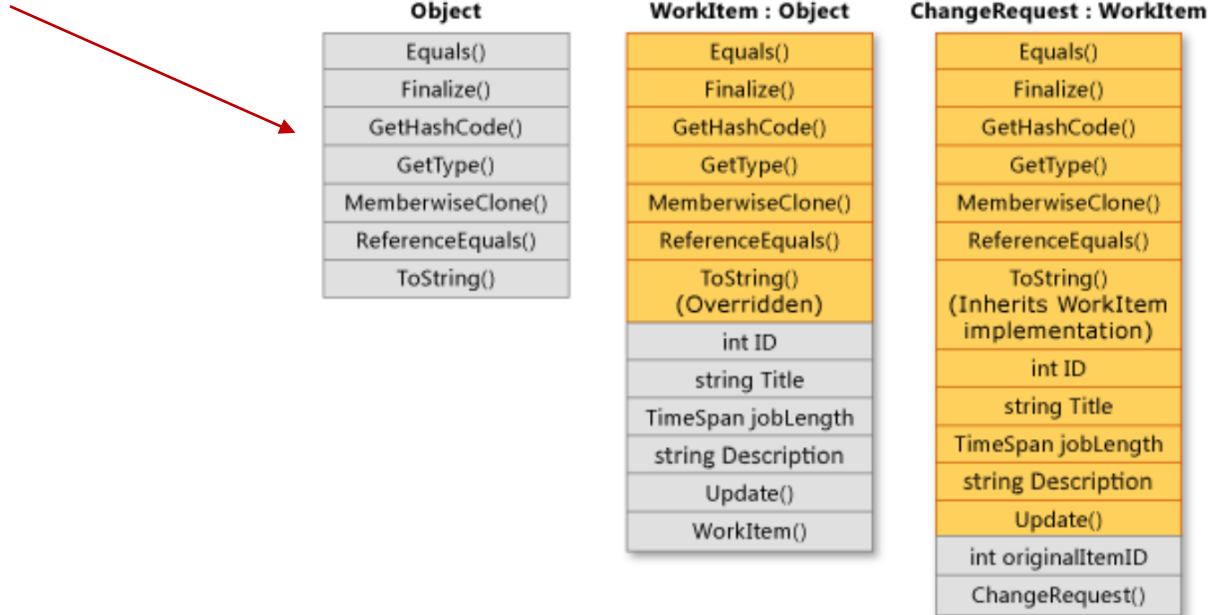
Declaration And Use II

- » sub-class can be used where superclass
 - **Implicit** upcast
 - Downcasts must be performed explicitly
- » Downcasting or type refinement is the act of casting a base class object to one of its derived classes

```
public class Painter {  
    public static void Paint(Figure fig)  
    {  
        // ...  
    }  
}  
  
Rectangle myRect = new Rectangle();  
Painter.paint(myRect);  
  
Circle myCircle = new Circle();  
Painter.paint(myCircle);  
  
Figure myFigure = new Circle();  
Circle aCircle = (Circle) myFigure;
```

System.Object

- » All classes inherit from System.Object class
- » Provides specific methods
- » Provides methods:
 - Equals()
 - GetType()
 - ToString()
 - GetHashCode()



Virtual Methods – override

- » A base method marked as virtual can be overridden by derived classes
- » Keyword: **override**
- » Used to provide a specialized implementation
- » If casted to base again, will invoke the override the base method
- » Not using **override** will issue a warning (not an error though)...

```
public class Figure {  
    public virtual void Output() {  
        Console.WriteLine("Figure object");  
    }  
}  
  
public class Circle : Figure {  
    public override void Output() {  
        Console.WriteLine("Circle object");  
    }  
}  
  
public class Rectangle : Figure {  
    public override void Output() {  
        Console.WriteLine("Rectangle object");  
    }  
}
```


Virtual Methods – new

- » hide base class members
- » **new** to suppress compiler warning
- » Not often used in real code
- » If casted to base, will invoke the base method
- » Typically, **override** is preferred!

```
public class Figure {  
    // ...  
    public void Output() {  
        Console.WriteLine("Base class");  
    }  
}  
  
public class Circle : Figure {  
    // ...  
    public new void Output() {  
        Console.WriteLine("Circle class");  
    }  
}
```

[Live coding](#)

41

The Keyword Base

- » Similar to keyword `this`
- » Used to access an overridden method from the base class
- » Used to call a base class constructor

```
public class Figure {  
    public virtual void Output() {  
        Console.WriteLine("Figure object");  
    }  
}  
  
public class Circle : Figure {  
    public override void Output() {  
        base.Output();  
        Console.WriteLine("Circle object");  
    }  
}  
  
// outputs:  
Circle circle = new Circle();  
circle.Output(); // prints "Figure object\nCircle object"
```

Abstract classes

```
public abstract class Figure {  
    public abstract double Area();  
}  
  
public class Circle : Figure {  
    public override double Area() {  
        return Math.Pi * Radius * Radius;  
    }  
}
```

- » Basically “concept” classes
- » Can be used to indicate missing components or implementations
- » Abstract classes can’t be instantiated (no objects!!!)
- » Abstract methods can only be contained in abstract classes
- » All abstract methods must be implemented by the child class

Summary

- » **virtual**: indicates that a method may be overridden by an inheritor
- » **override**: overrides the functionality of a virtual method in a base class, providing different functionality.
- » **new**: hides the original method (which doesn't have to be virtual), providing different functionality. This should only be used where it is absolutely necessary.
- » **abstract**: abstract methods must be implemented by the child and don't contain a body. Abstract classes can only be inherited, never instantiated!



THANK YOU!