Design Patterns

Gerwin van Dijken (gerwin.vandijken@inholland.nl)

Material

Moodle-course:

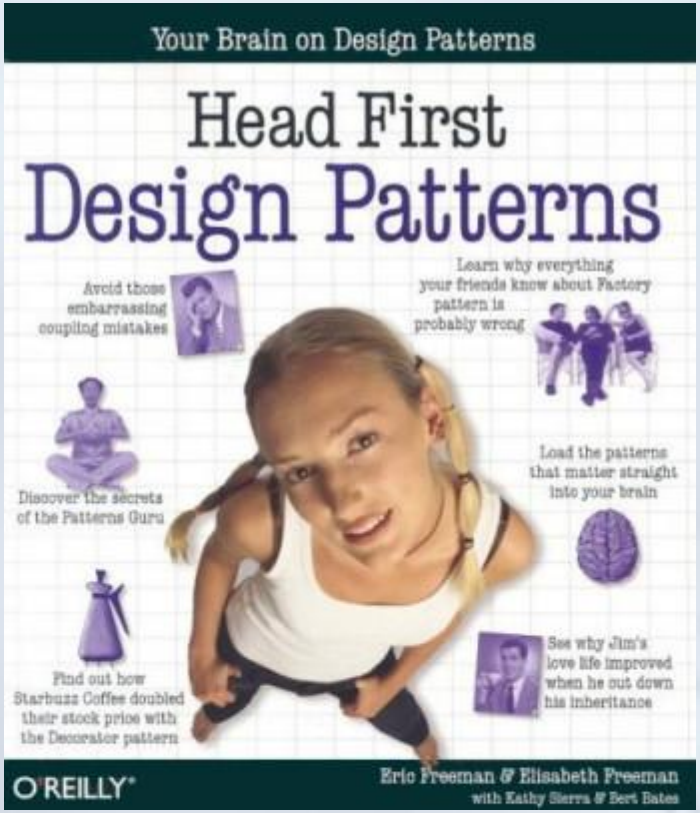
- 1920 IT1.4 Development

Weekly assignments:

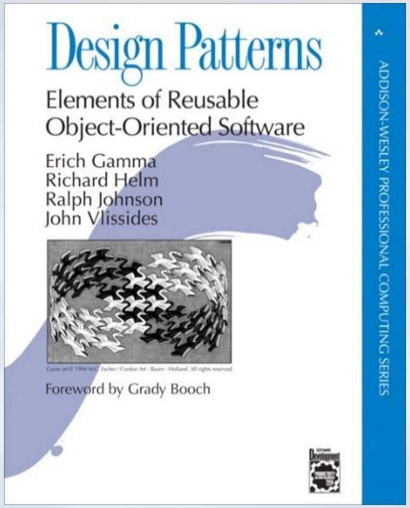
- can be found on Moodle
- mandatory
- help/feedback during consultancy meetings *(MS Teams)*
- upload on Moodle *(deadline always the next week)*
- checking off is done by lecturer *(offline, not in class)*

Recommended book (not required)

- 'Head First Design Patterns' (Java…)


Your Brain on Design Patterns
Head First
Design Patterns
Avoid those embarrassing coupling mistakes
Learn why everything your friends know about Factory pattern is probably wrong
Discover the secrets of the Patterns Guru
Load the patterns that matter straight into your brain
Find out how Starbuzz Coffee doubled their stock price with the Decorator pattern
See why Jim's love life improved when he cut down his inheritance
O'REILLY
Eric Freeman & Elisabeth Freeman
with Kathy Sierra & Bert Bates

GoF


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Program term 1.4

01 (wk-15)	abstract classes and interfaces
02 (wk-16)	Template Method pattern / Observer pattern
03 (wk-17)	MVC pattern
04 (wk-18)	*no classes*
05 (wk-19)	Strategy pattern / Adapter pattern
06 (wk-20)	Singleton pattern / State pattern
07 (wk-21)	Factory patterns
08 (wk-22)	repetition / practice exam

----

09 (wk-23)	exam *(computer assignments)*
10 (wk-24)	*retakes (courses term 1.3)*
11 (wk-25)	*retakes (courses term 1.4)*

Summary of Programming 3

- In term 1.3 the following topics were discussed:
  - classes: contain data and code/functionality
  - derived classes: derive from a base class
  - members: data and methods of an object
  - properties (set and get): member fields with access control, could be read-only
  - access modifiers: accessibility of members (public, protected, private)

Member fields are by default private or protected; we can make them accessible through public properties.

What are design patterns?

- A design pattern is a pattern for common problems (‘invented’ by others)

- It is meant to make software:
  - better maintainable;
  - more suitable for extensions/adjustments;
  - more flexible;

Design patterns are not suitable for small applications; they are only meaningful for large/complex software applications.

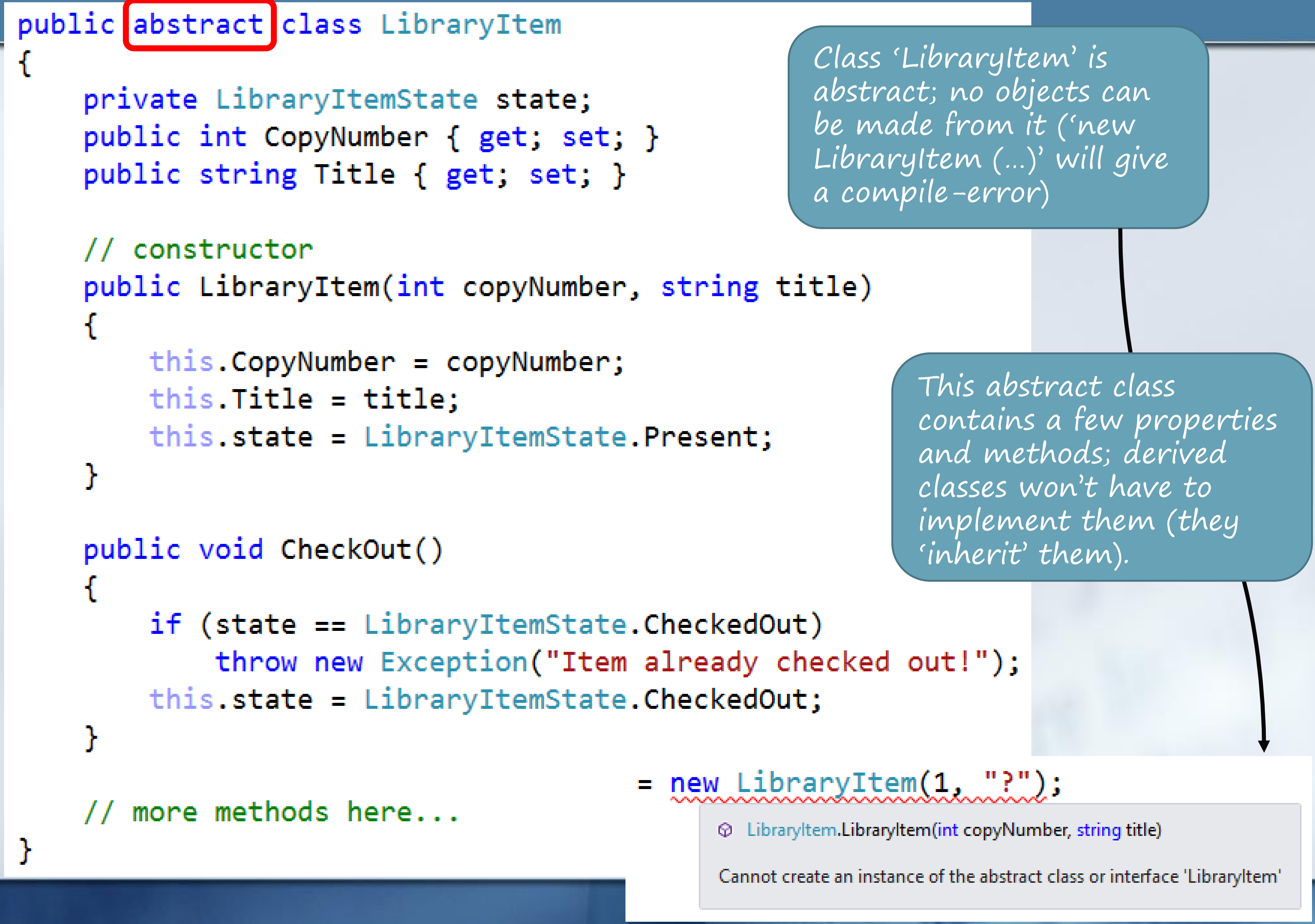
Interfaces

- The foundation for most Design Patterns are ‘interfaces’
- Let’s see what’s the difference between an abstract class and an interface…

Abstract classes

- Are always the base for other classes (base classes can not be instantiated: no objects can be made)
- Contain partial implementation (data and/or methods)
- Can have one or more methods without body; derived classes must implement these (abstract) methods

Example of an abstract class



```
public abstract class LibraryItem
{
    private LibraryItemState state;
    public int CopyNumber { get; set; }
    public string Title { get; set; }

    // constructor
    public LibraryItem(int copyNumber, string title)
    {
        this.CopyNumber = copyNumber;
        this.Title = title;
        this.state = LibraryItemState.Present;
    }

    public void CheckOut()
    {
        if (state == LibraryItemState.CheckedOut)
            throw new Exception("Item already checked out!");
        this.state = LibraryItemState.CheckedOut;
    }

    // more methods here...
}
```

Class 'LibraryItem' is abstract; no objects can be made from it ('new LibraryItem (...)' will give a compile-error)

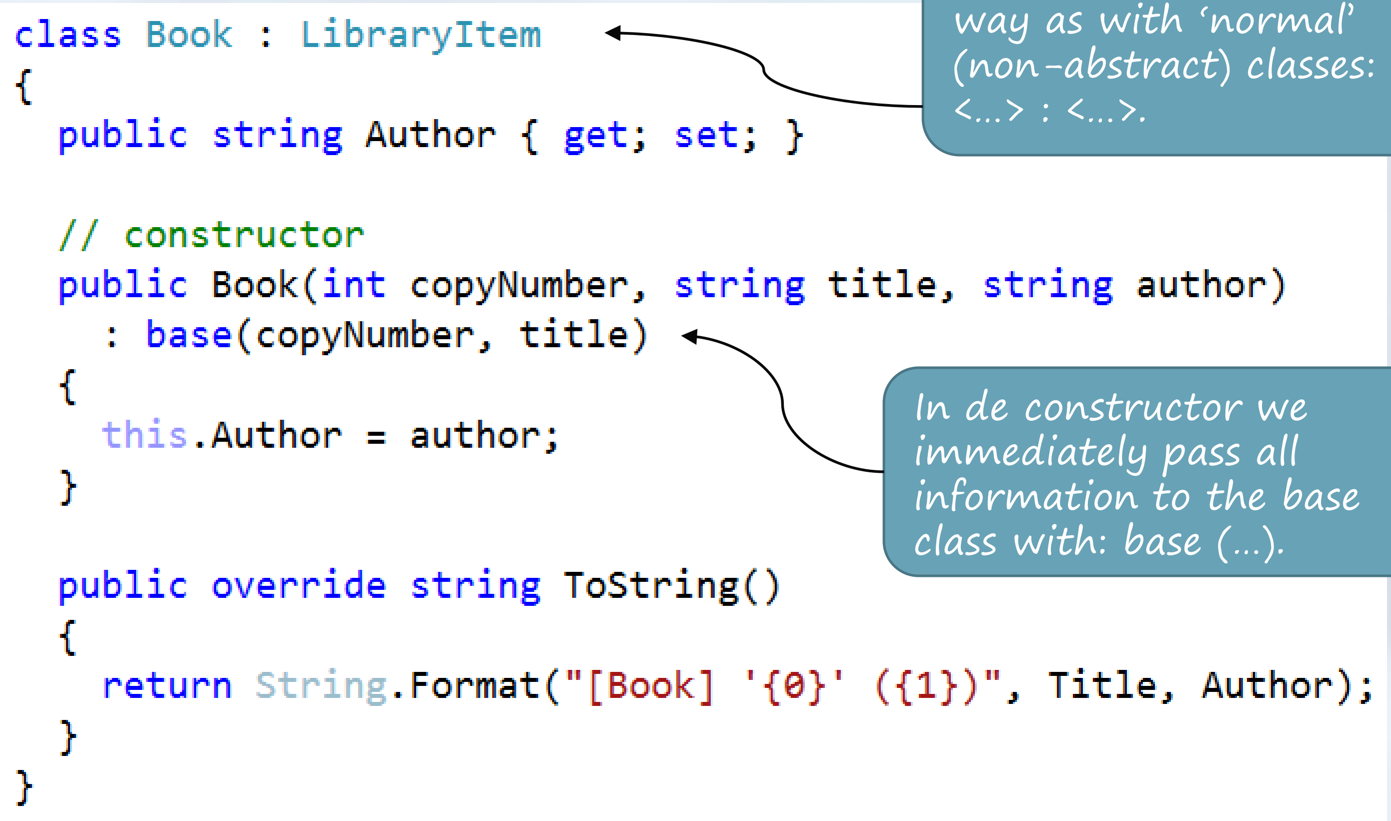
This abstract class contains a few properties and methods; derived classes won't have to implement them (they 'inherit' them).

```
= new LibraryItem(1, "?");
```

LibraryItem.LibraryItem(int copyNumber, string title)

Cannot create an instance of the abstract class or interface 'LibraryItem'

A derived class



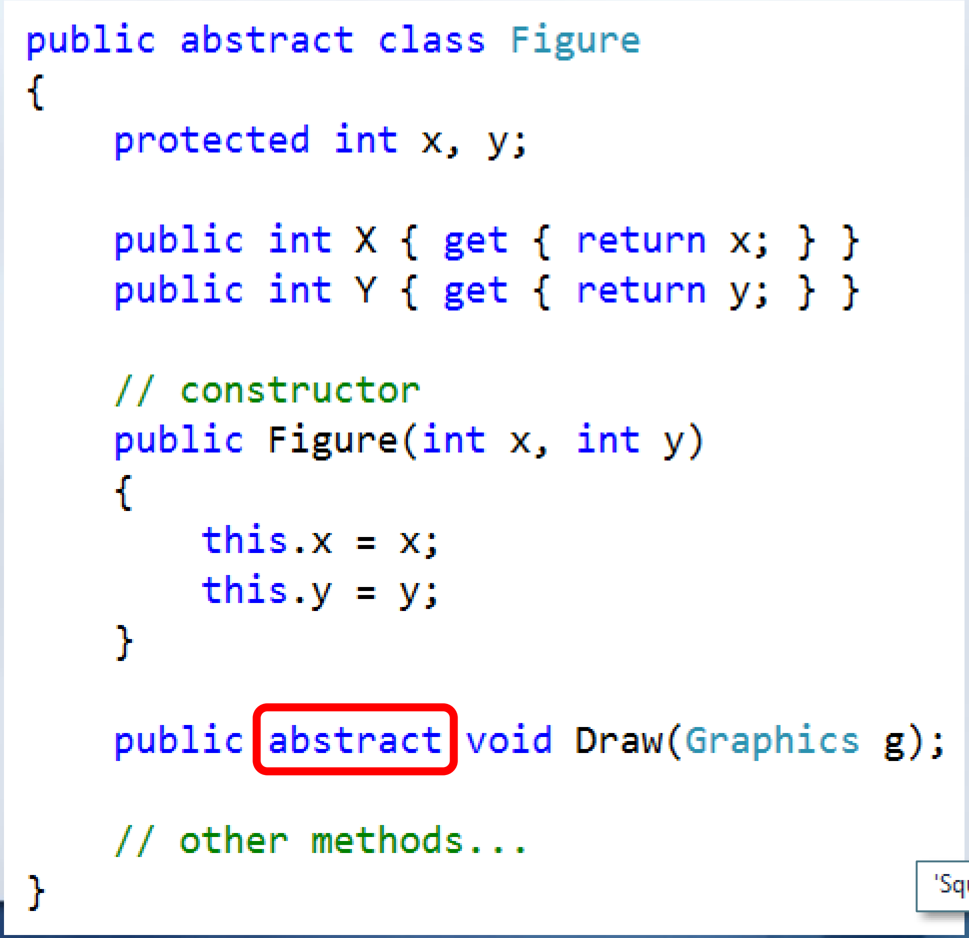
```
class Book : LibraryItem
{
  public string Author { get; set; }

  // constructor
  public Book(int copyNumber, string title, string author)
    : base(copyNumber, title)
  {
    this.Author = author;
  }

  public override string ToString()
  {
    return String.Format("[Book] '{0}' ({1})", Title, Author);
  }
}
```

Deriving is done the same way as with 'normal' (non-abstract) classes: <...> : <...>.

In de constructor we immediately pass all information to the base class with: base (...).

Abstract methods

- Abstract methods must be implemented by a derived class

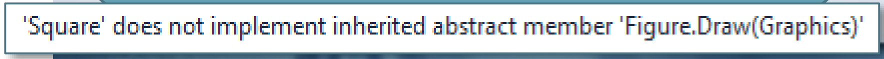


```
public abstract class Figure
{
    protected int x, y;

    public int X { get { return x; } }
    public int Y { get { return y; } }

    // constructor
    public Figure(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    public abstract void Draw(Graphics g);

    // other methods...
}
```

Class ‘Figure’ is abstract; no objects can be created with: new Figure(…)

A derived class must implement method ‘Draw’, since this method is abstract.

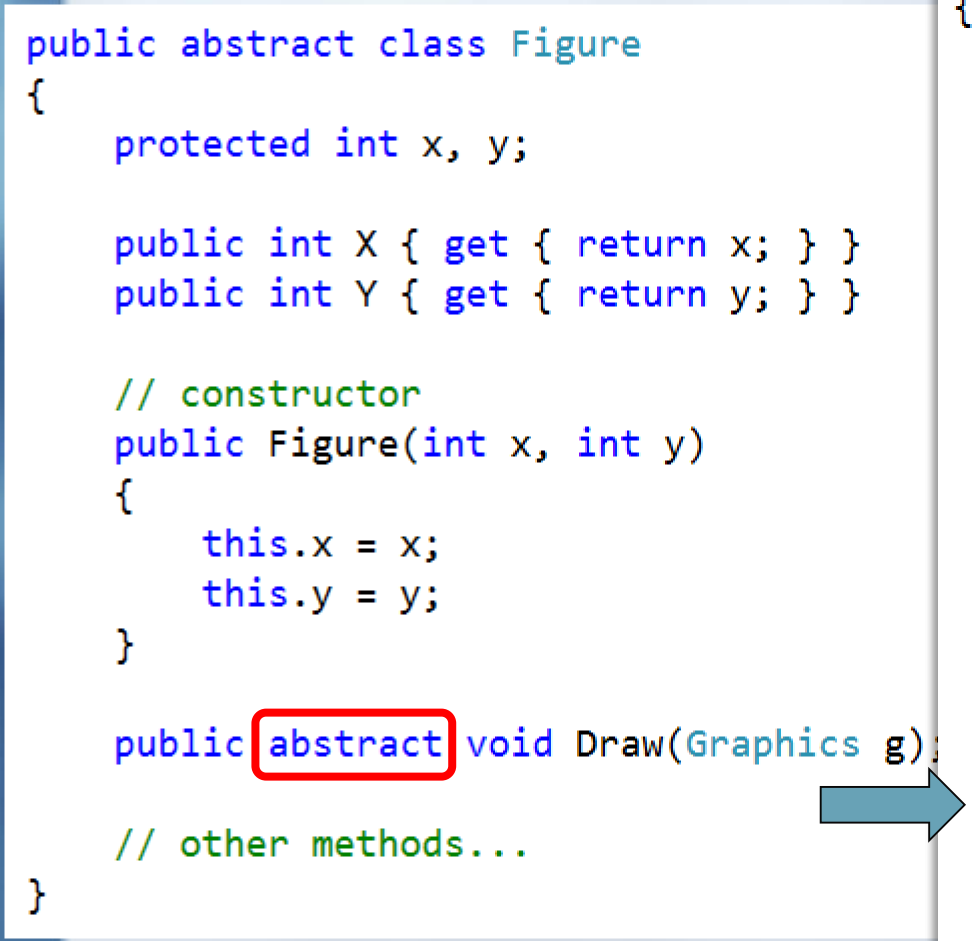
If an abstract method is not implemented, then a compile-error occurs:
‘<class-name> does not implement inherited abstract member <method-name>’.



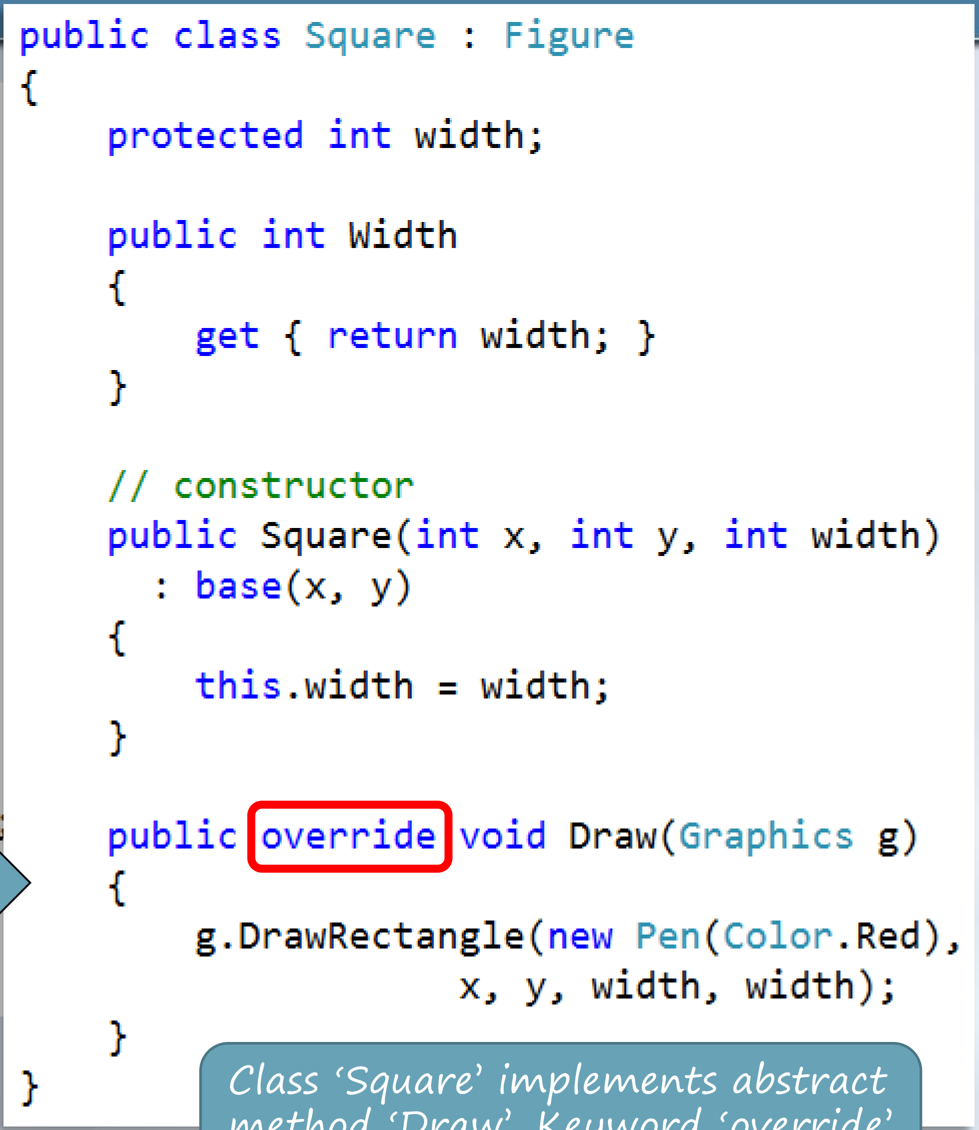
'Square' does not implement inherited abstract member 'Figure.Draw(Graphics)'

Abstract methods

Class ‘Square’ derives from abstract class ‘Figure’.



```
public abstract class Figure
{
    protected int x, y;

    public int X { get { return x; } }
    public int Y { get { return y; } }

    // constructor
    public Figure(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    public abstract void Draw(Graphics g);

    // other methods...
}
```



```
public class Square : Figure
{
    protected int width;

    public int Width
    {
        get { return width; }
    }

    // constructor
    public Square(int x, int y, int width)
      : base(x, y)
    {
        this.width = width;
    }

    public override void Draw(Graphics g)
    {
        g.DrawRectangle(new Pen(Color.Red),
                    x, y, width, width);
    }
}
```

Class ‘Square’ implements abstract method ‘Draw’. Keyword ‘override’ is needed here.

Interfaces

An interface has no implementation (no data and no implemented methods).

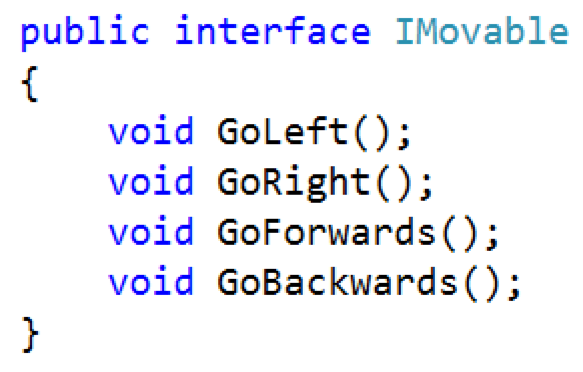
- A disadvantage of abstract classes is that a class can only derive from one base class
- A class can “implement” multiple interfaces
- An interface can be seen as a contract, ‘sub-classes’ must comply with this contract
- An interface contains only abstract methods/properties *(no implementation!)*

Example of an interface

- An interface describes behaviour, that other classes have to implement (Movable, Sortable, Comparable, …)

Methods in an interface are empty, there's no code between brackets { … }, only a ';'. The methods are public 'by default'.

Good practice: always use a capital 'I' for an interface name.



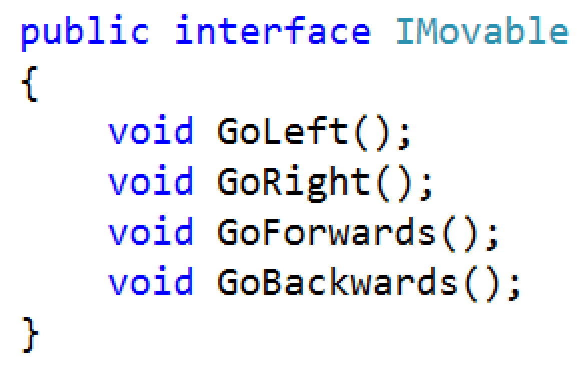
```
public interface IMovable
{
    void GoLeft();
    void GoRight();
    void GoForwards();
    void GoBackwards();
}
```

'Movable' things can move to 4 directions.

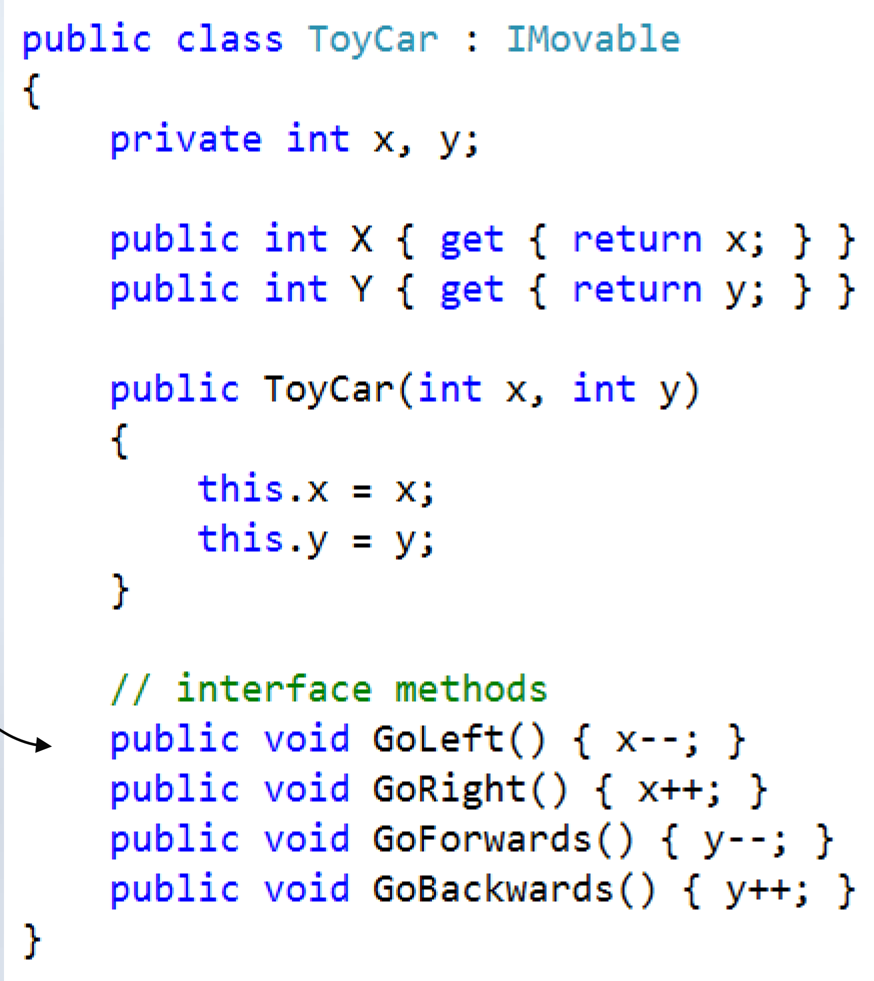
Classes that implement an interface, must implement all interface methods (in the given example 4 methods), with exactly the same signature (name and parameters).

Example of an interface

Implementing an interface is done the same as deriving from a class: <…> : <…>.



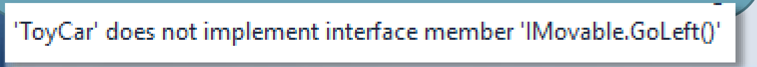
```
public interface IMovable
{
    void GoLeft();
    void GoRight();
    void GoForwards();
    void GoBackwards();
}
```



```
public class ToyCar : IMovable
{
    private int x, y;

    public int X { get { return x; } }
    public int Y { get { return y; } }

    public ToyCar(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    // interface methods
    public void GoLeft() { x--; }
    public void GoRight() { x++; }
    public void GoForwards() { y--; }
    public void GoBackwards() { y++; }
}
```

Class 'ToyCar' implements interface 'IMoveable' by implementing the 4 methods.

If an interface method is not implemented, a compile-error occurs: '<class-name> does not implement interface member <method-name>'.

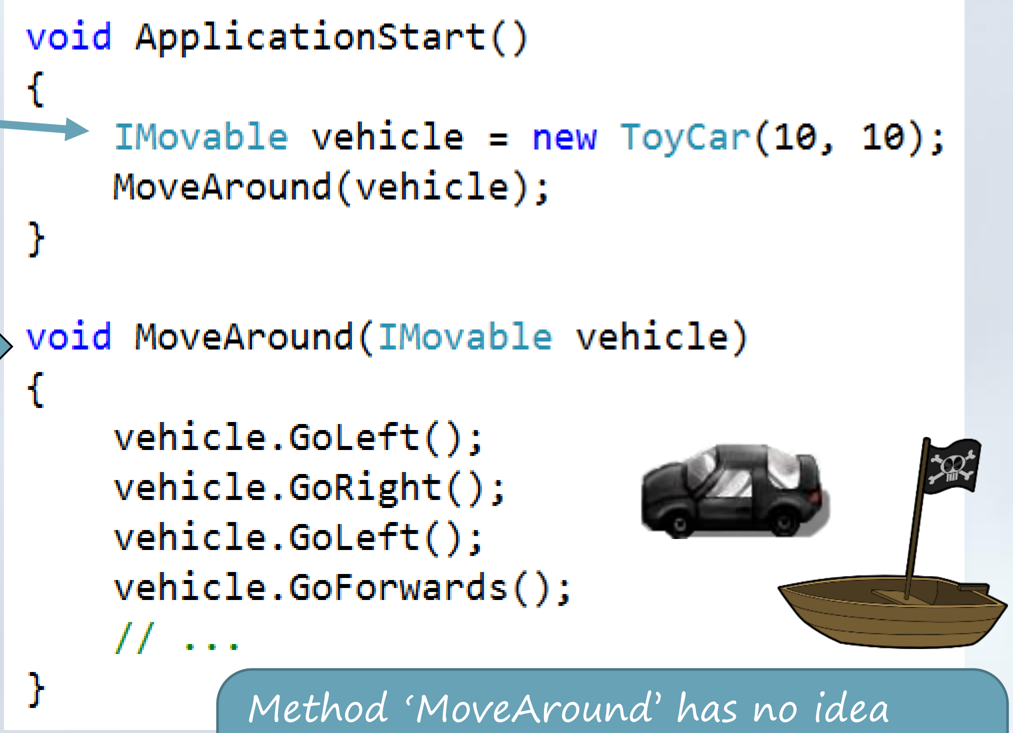


'ToyCar' does not implement interface member 'IMovable.GoLeft()'

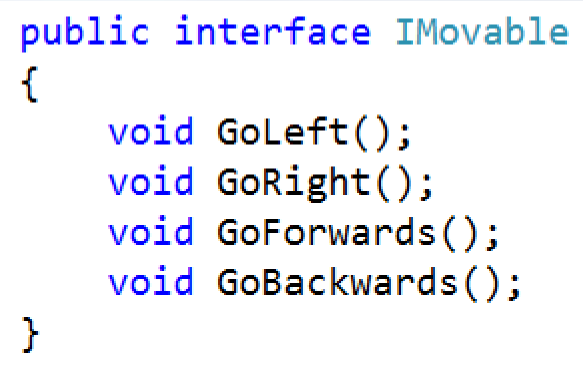
Example of an interface

- We can now program ‘against an interface’, so the code doesn’t need to know the real object/implementation!

We create a ‘ToyCar’, but the reference is of type ‘IMoveable’.



```
void ApplicationStart()
{
    IMovable vehicle = new ToyCar(10, 10);
    MoveAround(vehicle);
}

void MoveAround(IMovable vehicle)
{
    vehicle.GoLeft();
    vehicle.GoRight();
    vehicle.GoLeft();
    vehicle.GoForwards();
    // ...
}
```



```
public interface IMovable
{
    void GoLeft();
    void GoRight();
    void GoForwards();
    void GoBackwards();
}
```

Method ‘MoveAround’ has no idea what kind of vehicle is being moved, but it doesn’t care! (as long as it is ‘movable’…)

Summary

- No objects can be made *(with new)* from an abstract class
- Abstract methods in an abstract class must be implemented in the derived classes

- An interface contains only ‘abstract methods’
- A class can be derived from only one base class
- A class can implement multiple interfaces

- It is good practice to program ‘against an interface’

Assignments

- Moodle: ‘Week 1 assignments’