

SWEN20003
Object Oriented Software Development

Inheritance and Polymorphism - 2

Bach Le
bach.le@unimelb.edu.au

University of Melbourne
© University of Melbourne 2023

The Road So Far

- Subject Introduction
- Java Introduction
- Classes and Objects
- Software Tools and Bagel
- Arrays and Strings
- Input and Output
- Inheritance and Polymorphism - 1

Learning Outcomes

Upon completion of this topic you will be able to:

- Use **inheritance** to abstract common properties of classes
- Explain the relationship between a **superclass** and a **subclass**
- Make better use of **privacy** and **information hiding**
- Identify errors caused by **shadowing** and **privacy leaks**, and avoid them
- Describe and use method **overriding**
- Describe the **Object** class, and the properties inherited from it
- Describe what **upcasting** and **downcasting** are, and when they would be used
- Explain **polymorphism**, and how it is used in Java
- Describe the purpose and meaning of an **abstract** class

Overview

This topic will be delivered through two lectures (Lectures 9 and 10) each covering the following subtopics:

Lecture 9:

- Introduction and Motivation
- Inheriting Attributes
- Inheriting and Overriding methods

Lecture 10:

- Inheritance and Information Hiding
- The Object Class
- Abstract Classes

From the Previous Lecture

Extension Through Overriding - A Better Design

```
1      public class Piece {
2          private int currentRow;
3          private int currentColumn;
4          // Constructors getters and setters as before, not shown here
5
6          public boolean move(int toRow, int toColumn) {
7              System.out.println("Piece class: move() method");
8              if (!isValidMove(toRow, toColumn))
9                  return false;
10             this.currentRow = toRow;
11             this.currentColumn = toColumn;
12             return true;
13         }
14
15         public boolean isValidMove(int toRow, int toColumn) {
16             System.out.println("Piece class: isValidMove() method");
17             return toRow >= 0 && toRow < BOARD_SIZE &&
18                    toColumn >= 0 && toColumn < BOARD_SIZE;
19         }
20     }
```

Extension Through Overriding - A Better Design

```
1      public class Rook extends Piece {
2          ...
3          public boolean isValidMove(int toRow, int toColumn) {
4              boolean isValid = true;
5              System.out.println("Rook class: isValidMove() method");
6              if (!super.isValidMove(toRow, toColumn))
7                  return false;
8              //Logic for checking valid move and set isValid Rook
9              return isValid;
10         }
11     }
```

```
1      public class Knight extends Piece {
2          ...
3          public boolean isValidMove(int toRow, int toColumn) {
4              boolean isValid = true;
5              System.out.println("Rook class: isValidMove() method");
6              if (!super.isValidMove(toRow, toColumn))
7                  return false;
8              //Logic for checking valid move and set isValid for Knight
9              return isValid;
```

Testing Method Inheritance

```
1      public class InheritanceTester {
2          public static void main(String[] args) {
3              Rook rook1 = new Rook(2, 4);
4              rook1.move(4,10);
5              System.out.println("rook1 location: " + rook1);
6              System.out.println();
7
8              Piece rook2 = new Rook(3,5);
9              rook2.move (4,4);
10             System.out.println("rook2 location: " + rook2);
11             System.out.println();
12
13             Piece rook3 = new Piece(4,6);
14             rook3.move(8,12);
15             System.out.println("rook3 location: " + rook3);
16         }
17     }
```


Testing Method Inheritance

Program Output:

```
Piece class: move() method
Rook class: isValidMove() method
Piece class: isValidMove() method
rook1 location: (2,4)
```

```
Piece class: move() method
Rook class: isValidMove() method
Piece class: isValidMove() method
rook2 location: (4,4)
```

```
Piece class: move() method
Piece class: isValidMove() method
rook3 location: (4,6)
```

Inheritance and Information Hiding

Pitfall: Method Overriding

`private` methods cannot be overridden.

```
1 public class Piece {  
2     private boolean isValidMove(int currentRow, int currentColumn {... }  
3 }
```

```
1 public class Rook extends Piece {  
2     @Override  
3     private boolean isValidMove(int currentRow, int currentColumn) { ..}  
4 }
```

The above definition of the Rook is not valid.

```
1 public class Rook extends Piece {  
2     private boolean isValidMove(int currentRow, int currentColumn) { .. }  
3 }
```

The second definition of the Rook is valid, but does not override the `isValidMove()` method in the `Piece` class - will not get called from a parent class reference and cannot call the parent method with keyword `super`

Restricting Inheritance

If you don't want subclasses to override a method, you can use **final**!

Keyword

final: Indicates that an **attribute**, **method**, or **class** can only be assigned, declared or defined once.

Restricting Inheritance

Keyword

final: Final methods may not be overridden by subclasses.

```
1  public class Piece {  
2  
3      public final boolean move(int toRow, int toColumn) {  
4          System.out.println("Piece class: move() method");  
5          if (!isValidMove(toRow, toColumn))  
6              return false;  
7          this.currentRow = toRow;  
8          this.currentColumn = toColumn;  
9          return true;  
10     }  
11 }
```

This will restrict the `move()` method being overridden.

Access Control

Child classes **cannot** call **private** methods, and **cannot** access **private** attributes of parent classes.

```
1 public class Piece {  
2     private int currentRow;  
3     private int currentColumn;  
4     // Other methods go here  
5 }
```

```
1 public class Rook extends Piece {  
2     public int getCurrentRow() {  
3         return this.currentRow;  
4     }  
5     public void setCurrentRow(int currentRow) {  
6         this.currentRow = currentRow;  
7     }  
8 }
```

The above code for the Rook class is not valid.

Access Control

Child classes **can** call **protected** methods, and **can** access **protected** attributes of parent classes.

```
1 public class Piece {  
2     protected int currentRow;  
3     protected int currentColumn;  
4     // Other methods go here  
5 }
```

```
1 public class Rook extends Piece {  
2     public int getCurrentRow() {  
3         return this.currentRow;  
4     }  
5     public void setCurrentRow(int currentRow) {  
6         this.currentRow = currentRow;  
7     }  
8 }
```

The above code for the Rook class is valid, but see next slide!

Privacy Leaks

Defining attributes as `protected` allows updating them directly from child classes.

However, this should be avoided because it results in **privacy leaks**. The attributes of the parent class should be accessed via `public` or `protected` methods in the parent class.

Example:

- A good design of the `Piece` class should ensure that any method that updates the attributes, `currentRow`, `currentColumn`, checks if the new position is valid.
- If the attributes are defined as `protected`, the child classes will be able to update the attributes, without doing such checks (checks cannot be enforced by the parent class), resulting in invalid states for the object.

Access Control

Methods in the parent class that are only used by subclasses should be defined as **protected**.

Let us revisit our design for the Piece class.

```
1  public class Piece {
2      private int currentRow;
3      private int currentColumn;
4      final static int BOARD_SIZE = 8;
5      public Piece(int currentRow, int currentColumn) {...}
6
7      public int getCurrentRow() {...}
8      public void setCurrentRow(int currentRow) {...}
9      public int getCurrentColumn() {...}
10     public void setCurrentColumn(int currentColumn) {...}
11
12     public final boolean move(int toRow, int toColumn) {...}
13     public boolean isValidMove(int toRow, int toColumn) {...}
14     public String toString() {...}
15 }
```

Is there any method that should be defined as **protected**?

Access Control

The `isValidMove()` method in the `Piece` class is better defined as **protected** because:

- it should be accessed by the child class; and
- should not be accessed directly because the logic is not complete.

```
1  public class Piece {
2      private int currentRow;
3      private int currentColumn;
4      final static int BOARD_SIZE = 8;
5      public Piece(int currentRow, int currentColumn) {...}
6
7      public int getCurrentRow() {...}
8      public void setCurrentRow(int currentRow) {...}
9      public int getCurrentColumn() {...}
10     public void setCurrentColumn(int currentColumn) {...}
11
12     public final boolean move(int toRow, int toColumn) {...}
13     protected boolean isValidMove(int toRow, int toColumn) {...}
14     public String toString() {...}
15 }
16
```

Access Control

When overriding a method, a child class cannot further restrict the visibility of an overridden method. When overriding:

- a `public` method in the parent class must remain `public` in the child class
- a `protected` method in the parent class can remain `protected` in the derived or can be made `public`
- a `private` method in the parent class cannot be overridden - as discussed before

Inheritance and Shadowing -Example

```
1  public class PieceS {
2      public int currentRow;
3      public int currentColumn;
4      final static int BOARD_SIZE = 8;
5      public PieceS(int currentRow, int currentColumn) {
6          this.currentRow = currentRow;
7          this.currentColumn = currentColumn;
8      }
9      public int getCurrentRow() { return this.currentRow;}
10     ...
11 }
```

```
1  public class RookS extends PieceS {
2      public int currentRow;
3      public int currentColumn;
4      public RookS(int currentRow, int currentColumn) {
5          super(currentRow, currentColumn);
6      }
7      public int getCurrentRow() { return this.currentRow;}
8      ...
9  }
```

Inheritance and Shadowing -Example

```
1  public class DemoShadowing {
2      public static void main(String[] args) {
3          RookS r1 = new RookS(4,3);
4          System.out.println("r1: row print 1: " + r1.getCurrentRow());
5          System.out.println("r1: row print 2: "+ r1.currentRow);
6
7          PieceS r2 = new RookS(4,3);
8          System.out.println("r2: row print 1: " + r2.getCurrentRow());
9          System.out.println("r2: row print 2: " + r2.currentRow);
10     }
11 }
```

Program Output:

```
r1: row print 1: 0
r1: row print 2: 0
r2: row print 1: 0
r2: row print 2: 4
```

Inheritance and Shadowing

Keyword

Shadowing: When two or more variables are declared with the same name in **overlapping scopes**; for example, in both a subclass and superclass. The variable accessed will depend on the reference type rather than the object.

Don't. Do. It.

You only need to define (common) variables in the superclass.

Privacy Revisited

Keyword

public: Keyword when applied to a class, method or attribute makes it available/visible everywhere (within the class and outside the class).

Keyword

private: Keyword when applied to a method or attribute of a class, makes them only visible within that class. Private methods and attributes are not visible within *subclasses*, and are not inherited.

Keyword

protected: Keyword when applied to a method or attribute of a class, makes them only visible within that class, *subclasses* and also within all classes that are in the same package as that class. They are also visible to *subclasses* in other packages.

Visibility Modifiers

Modifier	Class	Package	Subclass	Outside
public	Y	Y	Y	Y
protected	Y	Y	Y	N
default	Y	Y	N	N
private	Y	N	N	N

Assess Yourself

Armed with these tools, how would you implement the Board class?

Assess Yourself

```
1  public class Board {
2      private Piece[] [] board;
3
4      public Board() {
5          // Initialize pieces with the correct locations
6          board[0][0] = new King(0,5);
7          board[0][1] = new Pawn(3,4);
8          //Code to place all pieces on the board - they should have the colour a
9          // example we have removed it to keep things simple
10     }
11
12     public boolean makeMove(int fromcurrentRow, int fromcurrentColumn,
13                             int tocurrentRow, int tocurrentColumn) {
14         if (board[fromcurrentRow][fromcurrentColumn] == null) {
15             return false;
16         }
17
18         Piece movingPiece = board[fromcurrentRow][fromcurrentColumn];
19
20         if (movingPiece.move(tocurrentRow, tocurrentColumn)) {
21             board[fromcurrentRow][fromcurrentColumn] = null;
22             board[tocurrentRow][tocurrentColumn] = movingPiece;
23             return true;
24         } else {
```

Assess Yourself

Implement the text-based chess game! The initial state of the board should be:

```
1      |a|b|c|d|e|f|g|h|
2      -----
3      8|R|N|B|Q|K|B|N|R|
4      -----
5      7|P|P|P|P|P|P|P|P|
6      -----
7      6| | | | | | | |
8      -----
9      5| | | | | | | |
10     -----
11     4| | | | | | | |
12     -----
13     3| | | | | | | |
14     -----
15     2|P|P|P|P|P|P|P|P|
16     -----
17     1|R|N|B|Q|K|B|N|R|
```

Assess Yourself

What classes do you need to write, and where would you write code for the “visualisation” part of the game?

How can you write your solution so that it doesn't matter whether it is a *text-based* or *3D* game?

Assess Yourself

You are a software developer in a swarm robotics laboratory, developing software for *heterogenous* swarms, or swarms with *multiple types* of robots.

A swarm can be an arbitrary combination of ground and aerial robots. Ground robots can be wheeled, bipedal (two legs), or spider-like (many legs). Aerial vehicles can be rotary, or winged.

Create a class design for this scenario, including appropriate use of *inheritance*, with *shared* or *common* attributes/behaviour defined in a superclass, and *specific* behaviour defined in subclasses.

Assess Yourself

Class: Robot

- Attributes
 - ▶ position
 - ▶ orientation
 - ▶ batteryLevel
- Methods
 - ▶ move

Assess Yourself

Class: AerialRobot **extends** Robot

- Attributes
 - ▶ altitude

Assess Yourself

Class: RotaryRobot **extends** AerialRobot

- Attributes
 - ▶ numRotors
- Methods
 - ▶ move

Assess Yourself

Class: WingedRobot **extends** AerialRobot

- Attributes
 - ▶ isPushPlane
- Methods
 - ▶ move

Assess Yourself

And so on...

The Object Class

Object

Every class in Java implicitly inherits from the `Object` class

- All classes are of type `Object`
- All classes have a `toString` method
- All classes have an `equals` method
- ... among other (less important) things

The toString Method

Consider the Piece class without a toString() method.

```
1 public class TestInheritance {  
2  
3     public static void main(String[] args) {  
4         Piece rook1 = new Rook(3, 5);  
5         System.out.println("rook1 location: " + rook1);  
6     }  
7  
8 }
```

rook1 location: Rook@1540e19d

The inherited toString method is pretty useless, so we **override** it.

The toString Method

Adding a toString method to the Piece class.

```
1 public class Piece {
2     ...
3     @Override
4     public String toString() {
5         return "(" + currentRow + "," + currentColumn + ")";
6     }
7 }
```

```
1 public class TestInheritance {
2     public static void main(String[] args) {
3         Piece rook1 = new Rook(2, 4);
4         System.out.println("rook1 location: " + rook1);
5     }
6 }
```

```
1 rook1 location: (2,4)
```

The equals Method

If have not added an equals method to our Piece class or the Rook class so far.

However, we can still call it because it is defined in the Object class.

```
1 public class TestInheritance {
2     public static void main(String[] args) {
3         Piece rook1 = new Rook(2, 4);
4         Piece rook2 = new Rook(2, 4);
5         System.out.println(rook1.equals(rook2));
6     }
7 }
```

false

The inherited equals method is equally useless (returns false anyway), but **overriding** is a bit more work

The equals Method

What do you think the *signature* would be for equals?

```
1 public class Piece {  
2     public boolean equals(Piece otherPiece) {  
3         <block of code to execute>  
4     }  
5 }
```

Although this works, it really did not override the equals method in the `Object` class.

Remember that equals is inherited from the `Object` class has the following signature:

```
public boolean equals(Object otherObject)
```


The equals Method

Note: If you auto generated the code for the Piece class, using IntelliJ IDE you will get the following.

```
1  @Override
2  public boolean equals(Object o) {
3      if (this == o) return true;
4      if (o == null) return false;
5      if (getClass() != o.getClass()) return false;
6      Piece piece = (Piece) o;
7      return (currentRow == piece.currentRow &&
8              currentColumn == piece.currentColumn);
9  }
10 @Override
11 public int hashCode() {
12     return Objects.hash(currentRow, currentColumn);
13 }
```

This method overrides the equals method in the Object class and is the one you should be using - not really the one we introduced in the topic Classes and Objects! The logic can be replaced, depending on how you want to compare the objects.

Useful Terms

Keyword

getClass: Returns an object of type `Class` that represents the details of the *calling object's class*.

Keyword

instanceof: An *operator* that gives `true` if an object A is an instance of the same class as object B, or a class that inherits from B.

```
return new Rook() instanceof Piece; // true
return new Piece() instanceof Rook; // false
```

Useful Terms

Keyword

Upcasting: When an object of a *child* class is assigned to a variable of an *ancestor* class.

```
Piece p = new Rook(2,3);
```

Keyword

Downcasting: When an object of an *ancestor* class is assigned to a variable of a *child* class. Only makes sense if the underlying object is **actually** of that class. Why?

```
Piece robot = new WingedRobot();  
WingedRobot plane = (WingedRobot) robot;
```

Polymorphism

Keyword

Polymorphism: The ability to use objects or methods in many different ways; roughly means “multiple forms”.

Overloading same method with various forms depending on **signature**
(Ad Hoc polymorphism)

Overriding same method with various forms depending on **class**
(Subtype polymorphism)

Substitution using subclasses in place of superclasses (Subtype polymorphism)

Generics defining parametrised methods/classes (Parametric polymorphism, *coming soon*)

Abstract Classes

Assess Yourself

Is there anything *strange* about our Piece class we defined?

- What is a Piece?
- If we create a Piece object, what does that mean?
- Does it make sense to have an object of type Piece?

Abstract

How would this code work?

```
Piece p1 = new Piece();  
p1.move(...)
```

It doesn't!

Some classes aren't meant to be instantiated because they aren't **completely defined**.

Although they are nouns they do not correspond to a real-world entity but is only an **abstraction** to define a class of entities (in this example game pieces such as pawns, rooks etc.).

Abstract Classes

Keyword

Abstract Class: A class that represents common attributes and methods of its subclasses, but that is **missing** some information specific to its subclasses. Cannot be instantiated.

Keyword

Concrete Class: Any class that is not abstract, and has well-defined, specific implementations for all actions it can take.

Abstract Classes

Keyword

abstract: Defines a **class** that is **incomplete**. Abstract classes are “general concepts”, rather than being fully realised/detailed.

```
<visibility> abstract class <ClassName> {  
  
}
```

```
public abstract class Piece {  
    // Attributes and methods go here  
}
```

Abstract Methods

Keyword

abstract: Defines a superclass method that is common to **all** subclasses, but has no implementation. Each subclass then provides its own implementation through **overriding**.

```
<privacy> abstract <returnType> <methodName>(<arguments>);
```

```
public abstract boolean isValidMove(int toRow, int toColumn);
```

Note: If you make the `isValidMove()` method in the `Piece` class **abstract** (like above), it cannot have any implementation like what we did in our previous example. All the logic has to be implemented in the child classes.

Abstract vs. Concrete

Abstract classes are identical, except:

- **May** have abstract methods - abstract classes can have no abstract methods
- Classes with abstract methods **must** be abstract
- **Cannot** be instantiated
- Represent an **incomplete** concept, rather than a **thing** that is part of a problem

If a class is abstract:

```
public abstract class Piece {  
    // attribute and methods  
}
```

The following definition is not valid:

```
Piece p = new Piece(3,2); // Not valid
```

Using Abstract Classes Example

```
1      public abstract class Shape {
2          private double centreX;
3          private double centreY;
4
5          public Shape(double centreX, double centreY) {
6              this.centreX = centreX;
7              this.centreY = centreY;
8          }
9          // Getters setters go here - not shown
10
11
12      public abstract double computeArea();
13  }
```

Using Abstract Classes Example

```
1  public class Circle extends Shape {
2
3      private double radius;
4
5      public Circle(double centreX, double centreY, double radius) {
6          super(centreX, centreY);
7          this.radius = radius;
8      }
9
10     // Getters setters go here - not shown
11
12     @Override
13     public double computeArea() {
14         return Math.PI*Math.pow(radius,2.0);
15     }
16 }
```

Using Abstract Classes Example

```
1  public class Square extends Shape {
2
3      private double sideLength;
4
5      public Square(double centreX, double centreY, double sideLength){
6          super(centreX, centreY);
7          this.sideLength = sideLength;
8      }
9
10     // Getters setters go here - not shown
11
12     @Override
13     public double computeArea() {
14         return Math.pow(sideLength, 2.0);
15     }
16 }
```

Using Abstract Classes Example

```
1      public class TestAbstractShape {
2          public static void main(String[] args) {
3              Shape[] s = new Shape[3];
4              s[0] = new Circle(0,0, 1);
5              s[1] = new Square(0, 0, 2);
6              s[2] = new Square(0,0, 3);
7              computeShapeAreas(s);
8          }
9          public static void computeShapeAreas(Shape[] shapes) {
10             for (Shape s: shapes) {
11                 System.out.println("Area = " + s.computeArea());
12             }
13         }
14     }
15 }
```

Program Output:

Area = 3.141592653589793

Area = 4.0

Area = 9.0

Types of Inheritance

Inheritance can have multiple levels.

Example:

```
public abstract class Shape {  
    // Attributes and methods go here  
}
```

```
public class Circle extends Shape {  
    // Attributes and methods go here  
}
```

```
public class Rectangle extends Shape {  
    // Attributes and methods go here  
}
```

```
public class GraphicCircle extends Circle {  
    // Attributes and methods go here  
}
```


Types of Inheritance

More generally, there are different forms of inheritance.

- Single inheritance (only one super class)
- Multiple inheritance (several super classes)
- Hierarchical inheritance (one super class, many sub classes)
- Multi-Level inheritance (derived from a derived class)
- Hybrid inheritance (more than two types)
- Multi-path inheritance (inheritance of some properties from two sources).

Notes:

- Java does not support Multiple inheritance, and hence some other forms of inheritance that involves Multiple inheritance, such as Multi-path inheritance.
- Java *sort-of* supports multiple inheritance through Interfaces (our next topic), but it is not quite multiple inheritance

Assess Yourself

You are a software developer in a swarm robotics laboratory, developing software for *heterogenous* swarms, or swarms with *multiple types* of robots.

A swarm can be an arbitrary combination of ground and aerial robots. Ground robots can be wheeled, bipedal (two legs), or spider-like (many legs). Aerial vehicles can be rotary, or winged.

Create a class design for this scenario, including appropriate use of *inheritance*, with *shared* or *common* attributes/behaviour defined in a superclass, and *specific* behaviour defined in subclasses.

Assess Yourself

Class: Robot

- Attributes
 - ▶ position
 - ▶ orientation
 - ▶ batteryLevel
- Methods
 - ▶ move

Assess Yourself

Class: AerialRobot **extends** Robot

- Attributes
 - ▶ altitude

Assess Yourself

Class: RotaryRobot **extends** AerialRobot

- Attributes
 - ▶ numRotors
- Methods
 - ▶ move

Assess Yourself

Class: WingedRobot **extends** AerialRobot

- Attributes
 - ▶ isPushPlane
- Methods
 - ▶ move

Assess Yourself

```
public abstract class Robot {  
    public abstract void move(...);  
}
```

```
public abstract class AerialRobot extends Robot {  
  
}
```

```
public class WingedRobot extends AerialRobot {  
    public void move(...) {  
        <block of code to execute>  
    }  
}
```

Our Powers Combine!

Let's make some magic with our new tools...

Write a program that uses the Robot class and its descendants to create a swarm of robots, either using input from the user, or programmatically.

If we wanted to **control** this swarm, **interact** with it, or have it operate autonomously, what abstractions might we use?

Our Powers Combine!

```
import java.util.Random;

public class RobotApp {

    public static void main(String[] args) {
        final int MAX_ROBOTS = 10;
        Robot[] swarm = new Robot[MAX_ROBOTS];

        for (int i = 0; i < MAX_ROBOTS; i++) {
            swarm[i] = createRobot();
        }
        move(swarm);
    }
}
```

// Continued on next slide

Our Powers Combine!

```
private static Robot createRobot() {  
    final int NUM_ROBOT_TYPES = 1;  
    Random rand = new Random();  
  
    switch(rand.nextInt(NUM_ROBOT_TYPES)) {  
        case 0:  
            return new WingedRobot();  
        case 1:  
            return new RotaryRobot();  
    }  
}  
  
private static void move(Robot[] swarm) {  
    for (Robot r : swarm) {  
        r.move(new Point());  
    }  
}
```

Assess Yourself

As a developer for the local zoo, your team has been asked to develop an interactive program for the student groups who often come to visit.

The system allows users to search/filter through the zoo's animals to find their favourite, where they can then read details such as the animal's favourite food, where they're found, and other fun facts.

Your teammates are handling the user interface and mechanics; you've been tasked with designing the data representation that underpins the system.

Using what you know of Object Oriented Design, design a data representation for (some of) the animals in the zoo, making sure to use *inheritance* and *abstract classes* appropriately.

Learning Outcomes

Upon completion of this topic you will be able to:

- Use **inheritance** to abstract common properties of classes
- Explain the relationship between a **superclass** and a **subclass**
- Make better use of **privacy** and **information hiding**
- Identify errors caused by **shadowing** and **privacy leaks**, and avoid them
- Describe and use method **overriding**
- Describe the **Object** class, and the properties inherited from it
- Describe what **upcasting** and **downcasting** are, and when they would be used
- Explain **polymorphism**, and how it is used in Java
- Describe the purpose and meaning of an **abstract** class