SWEN20003
Object Oriented Software Development

Inheritance and Polymorphism - 1

Bach Le
bach.le@unimelb.edu.au

University of Melbourne

The Road So Far

- Subject Introduction
- A Quick Tour of Java
- Classes and Objects
- Software Tools and Bagel
- Arrays and Strings
- Input and Output

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

Introduction and Motivation

A Motivating Example

As a rookie game designer, you want to test your skills by implementing a simple, text-based game of chess.

Chess is a board game that has two players who take turns to move different types of pieces in the board based on the game rules - the goal is to capture the King.

What classes would you use, and what attributes and methods would they have?

Be sure to use **information hiding** and **access control**.

A Motivating Example

Class: `Chess (main)`

- Attributes
  - board
  - players
  - isWhiteTurn
- Methods
  - initialiseGame
  - isGameOver
  - getNextMove

A Motivating Example

Class: `Player`

- Attributes
  - colour
- Methods
  - makeMove

A Motivating Example

Class: `Board`

- Attributes
  - Pawn[]
  - Rook[]
  - Knight[]
  - Bishop[]
  - King
  - Queen
- Methods
  - getNextMove
  - isGameOver

A Motivating Example

Class: `Pawn`

- Attributes
  - isAlive
  - isWhite
  - currentRow
  - currentColumn
- Methods
  - move
  - isValidMove

A Motivating Example

Class: `Rook`

- Attributes
  - isAlive
  - isWhite
  - currentRow
  - currentColumn
- Methods
  - move
  - isValidMove

A Motivating Example

Class: `Bishop`

- Attributes
  - isAlive
  - isWhite
  - currentRow
  - currentColumn
- Methods
  - move
  - isValidMove

A Motivating Example

Class: `Knight`

- Attributes
  - isAlive
  - isWhite
  - currentRow
  - currentColumn
- Methods
  - move
  - isValidMove

A Motivating Example

Class: `Queen`

- Attributes
  - isAlive
  - isWhite
  - currentRow
  - currentColumn
- Methods
  - move
  - isValidMove

A Motivating Example

Class: `King`

- Attributes
  - isAlive
  - isWhite
  - currentRow
  - currentColumn
- Methods
  - move
  - isValidMove

What is the problem?

Why is the design for our Chess game poor?

- Repeated code/functionality, hard to debug
- Doesn't represent the "similarity"/relationship between the pieces
- A lot of work required to implement
- Difficult to extend

Pitfall: Poor Design

Think about how you might implement the `Board`...

```
public class Board {

    private Pawn[] pawns;
    private Rook[] rooks;
    ...



}
```

Pitfall: Poor Design

How might you implement methods for the game?

```
public void move(Pawn pawn) {
        ...
}

public void move(Rook rook) {
        ...
}

public void move(Knight knight) {
        ...
}
```

Most, if not all, of the code in these methods would be the same.
Are you a terrible programmer? No, you're just inexperienced, and have not learnt how to use Inheritance.

Inheritance

Keyword

Inheritance: A form of abstraction that permits "generalisation" of similar attributes/methods of classes; analogous to passing genetics on to your children.

Inheritance

Keyword

Superclass: The "parent" or "base" class in the inheritance relationship; provides general information to its "child" classes.

Keyword

Subclass: The "child" or "derived" class in the inheritance relationship; inherits common attributes and methods from the "parent" class.

Inheritance

- Define common attributes and methods in the Superclass (base class)
- Subclass automatically contains all (public/protected) instance variables and methods in the superclass
- Additional methods and/or instance variables can be defined in the subclass
- Inheritance allows code to be **reused**
- Subclasses should be "more specific" versions of a superclass

Designing Superclasses and Subclasses

How could we use inheritance in the chess game example?

What properties could be "generalised" across multiple classes?

Inheriting Attributes and Methods

How do we Inherit?

You can see that all attributes for the "Pieces", `Pawn`, `Rook`, `Knight`, `Bishop`, `King`, `Queen` are common or "general": `isAlive`, `isWhite`, `currentRow`, `currentColumn`.

So we can define them in a **parent class** (Superclass), named `Piece`, and all make all other pieces **child classes** (Subclasses) of the `Piece` class.

In the next example, I will only choose two attributes (`currentRow`, `currentColumn`) for demonstration purposes but the concepts can be used for any number of attributes.

Without Implementing Inheritance

```
public class Rook {
    private int currentRow;
    private int currentColumn;

    public int getCurrentRow() {
        return currentRow;
    }
    public void setCurrentRow(int currentRow) {
        this.currentRow = currentRow;
    }
    public int getCurrentColumn() {
        return currentColumn;
    }
    public void setCurrentColumn(int currentColumn) {
        this.currentColumn = currentColumn;
    }

    public void move(int toRow, int toColumn)  { .... }

    public boolean isValidMove(int toRow, int toColumn) { .... }
}
```

Without Implementing Inheritance

```
public class Knight {
    private int currentRow;
    private int currentColumn;

    public int getCurrentRow() {
        return currentRow;
    }
    public void setCurrentRow(int currentRow) {
        this.currentRow = currentRow;
    }
    public int getCurrentColumn() {
        return currentColumn;
    }
    public void setCurrentColumn(int currentColumn) {
        this.currentColumn = currentColumn;
    }

    public void move(int toRow, int toColumn)  { .... }

    public boolean isValidMove(int toRow, int toColumn) { .... }
}
```

Implementing Inheritance

Superclass

```
public class Piece {
    private int currentRow;
    private int currentColumn;

    public int getCurrentRow() {
        return currentRow;
    }
    public void setCurrentRow(int currentRow) {
        this.currentRow = currentRow;
    }
    public int getCurrentColumn() {
        return currentColumn;
    }
    public void setCurrentColumn(int currentColumn) {
        this.currentColumn = currentColumn;
    }

}
```

Implementing Inheritance

Subclasses

```
public class Rook extends Piece {
    public void move(int toRow, int toColumn)  { .... }

    public boolean isValidMove(int toRow, int toColumn) { .... }
}
```

```
 public class Knight extends Piece {
    public void move(int toRow, int toColumn) { ....}

    public boolean isValidMove(int toRow, int toColumn) {....}
 }
```

Both the `Rook` class and the `Knight` class inherit the attributes and the methods in the `Piece` class although they are not defined in the class itself.

But what does this really mean?

Defining Inheritance

Keyword

extends: Indicates one class **inherits** from another

- Inheritance defines an "**Is A**" relationship
  - All `Rook` objects are `Pieces`
  - All `Dog` objects are `Animals`
  - All `Husky` objects are `Dogs`
- Only use inheritance when this relationship **makes sense**
- A subclass can use attributes in the superclass - let us see how we do this

Creating Objects

```
1  public class InheritanceTester {
2      public static void main(String[] args) {
3          Rook rook1 = new Rook();
4          System.out.println("rook1 location: (" + rook1.getCurrentRow()
5                             "," + rook1.getCurrentColumn() + ")");
6
7          Piece rook2 = new Rook(); // Rook "is a" Piece
8          System.out.println("rook2 location: (" + rook2.getCurrentRow()
9                             "," + rook2.getCurrentColumn() + ")");
10
11         Rook rook3 = new Piece();//Invalid: a Piece "is not a" Rook
12     }
13 }
```

Program Output:

```
rook1 location: (0,0)
rook2 location: (0,0)
```

Although the getters and setters are in the parent class, the child class could use them, because they were *inherited* from the parent.

Initializing with Constructors

What about the Constructors?

Do we copy and paste parent constructors into subclass constructors?

Of course not!

The keyword `super` can be used to invoke (call) the constructor of the super class.

Keyword

super: Invokes a constructor in the **parent** class

Constructors

```java
public class Piece {
    private int currentRow;
    private int currentColumn;
    public Piece(int currentRow, int currentColumn) {
        this.currentRow = currentRow;
        this.currentColumn = currentColumn;
    }
    public int getCurrentRow() {..}
    public void setCurrentRow(int currentRow) {..}
    public int getCurrentColumn() {..}
    public void setCurrentColumn(int currentColumn) {..}
}
```

```java
public class Rook extends Piece {
    public Rook(int currentRow, int currentColumn) {
        super(currentRow, currentColumn);
        // Any other code
    }
    public void move(int toRow, int toColumn) {..}
    public boolean isValidMove(int toRow, int toColumn) {..}
}
```

Super Constructor

- May only be used within a subclass constructor
- Must be the first statement in the subclass constructor (if used)
- Parameter **types** to super constructor call must match that of the constructor in the base class

Initializing using Constructors

```
1  public class InheritanceTester {
2      public static void main(String[] args) {
3          Rook rook1 = new Rook(2, 10);
4          System.out.println("rook1 location: (" + rook1.getCurrentRow()
5              "," + rook1.getCurrentColumn() + ")");
6
7          Piece rook2 = new Rook(3, 5); // Rook "is a" Piece
8          System.out.println("rook2 location: (" + rook2.getCurrentRow()
9              "," + rook2.getCurrentColumn() + ")");
10
11         Rook rook3 = new Piece();//Invalid: Piece "is not a" Rook
12     }
13 }
```

Program Output:

```
rook1 location: (2,10)
rook2 location: (3,5)
```

Inheriting and Overriding Methods

How do we Inherit Methods?

Consider the two methods in our `Pieces`: `move` and `isValidMove`

If you consider the logic for implementing the `move()` method for the Pieces, what do you say?

- Regardless of the Piece, the logic is the same *if your code does not have to check if the new location is valid* (for now, let us assume somebody have validated the new location before calling the method)!

How about the `isValidMove()` method?

- All pieces must have this method, with the same signature.
- Some of the logic is common: e.g. checking if the new location is not outside the board.
- Some of the logic is different: e.g. the way a Rook can move is different to the way a Knight can move.

Implementing Method Inheritance

Now let us look at how we implement inheritance of methods, `move()` and `isValidMove()` methods.

```
1  public class Piece {
2      private int currentRow;
3      private int currentColumn;
4      // Getters and setters as before, not shown here
5      public void move(int toRow, int toColumn) {
6          System.out.println("Piece class: move() method");
7          this.currentRow = toRow;
8          this.currentColumn = toColumn;
9      }
10     public boolean isValidMove(int toRow, int toColumn) {
11         System.out.println("Piece class: isValidMove() method");
12         return true;
13     }
14     public String toString() {
15         return "(" + currentRow + "," + currentColumn + ")";
16     }
17 }
```

Implementing Method Inheritance

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

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

Testing Method Inheritance

```
1  public class InheritanceTester {
2      public static void main(String[] args) {
3          Rook rook1 = new Rook(2, 10);
4          if (rook1.isValidMove(4, 10))
5              rook1.move(4,10);
6          System.out.println("rook1 location: " + rook1);
7          System.out.println();
8
9          Piece rook2 = new Rook(3, 5);
10         if (rook2.isValidMove(6, 10))
11             rook2.move(6,10);
12         System.out.println("rook2 location: " + rook2);
13         System.out.println();
14
15         Piece rook3 = new Piece(4,6);
16         if (rook3.isValidMove(8, 12))
17             rook3.move(8,12);
18         System.out.println("rook3 location: " + rook3);
19     }
20 }
```

Testing Method Inheritance

Program Output:

```
1  Rook class: isValidMove() method
2  Piece class: move() method
3  rook1 location: (4,10)
4
5  Rook class: isValidMove() method
6  Piece class: move() method
7  rook2 location: (6,10)
8
9  Piece class: isValidMove() method
10 Piece class: move() method
11 rook3 location: (8,12)
```

Method Overriding

- When a method is defined only in the parent class (and has the correct visibility which we will discuss later), it gets called regardless of the type of object created (e.g. `move{}` method).
- When a method with the **same signature is defined both in the parent class and the child class**, which method executes purely depends on the **type of object** as opposed to the type of reference (e.g. `isValidMove()` method).
- In the latter case (method defined in both classes), the child class method **Overrides** the method in the parent class.
- Annotation @Override can be used in code to indicate that the method is overriding a method in the parent class (optional). See next slide for an example.

Implementing Overriden Methods

```
1
2   public class Rook extends Piece {
3
4       @Override
5       public boolean isValidMove(int toRow, int toColumn) {
6           boolean isValid = true;
7           System.out.println("Rook class: isValidMove() method");
8           // Logic for checking valid move and set isVaid
9           return isValid;
10      }
11
12  }
```

Note: IDEs support generation of code stubs for overridden methods, and they normally include the @Override annotation when generating such code stubs.

Method Overriding

Keyword

Overriding: Declaring a method that exists in a superclass **again** in a subclass, with the **same** signature. Methods can **only** be overriden by subclasses.

Keyword

Overloading: Declaring multiple methods with the same name, but **differing method signatures**. Superclass methods **can** be overloaded in subclasses.

Why Override?

- Subclasses can **extend** functionality from a parent
- Subclasses can **override/change** functionality of a parent
- Makes the *subclass* behaviour **available** when using references of the *superclass* type
- Defines a *general* "interface" in a superclass, with *specific* behaviour implemented in the subclass
  - This allows seamless access to methods in subclasses using a reference to the superclass - we will see examples later

Extension Through Overriding

Can you improve the design of the `isValidMove` method of your child classes (`Rook` and `Knight`)?

Remember, the logic had two parts:

- part that was common to all pieces - checking if the move is within the board
- part that is specific to a particular piece - checking if the move is valid for the particular type of piece

Can we move the generic logic to the parent class and re-use?

Extension Through Overriding - A Better Design

```java
public class Piece {
    final static int BOARD_SIZE = 8;
    ...
    public boolean isValidMove(int toRow, int toColumn) {
        System.out.println("Piece class: isValidMove() method");
        return toRow >= 0 && toRow < BOARD_SIZE &&
            toColumn >= 0 && toColumn < BOARD_SIZE;
    }
}
```

```java
public class Rook extends Piece {
    ...
    public boolean isValidMove(int toRow, int toColumn) {
        boolean isValid = true;
        System.out.println("Rook class: isValidMove() method");
        if (!super.isValidMove(toRow, toColumn))
            return false;
        //Logic for checking valid move and set isValid
        return isValid;
    }
}
```

Testing Method Inheritance

```
1       public class InheritanceTester {
2           public static void main(String[] args) {
3               Rook rook1 = new Rook(2, 4);
4               if (rook1.isValidMove(4, 10))
5                   rook1.move(4,10);
6               System.out.println("rook1 location: " + rook1);
7               System.out.println();
8
9               Piece rook2 = new Rook(3, 5);
10              if (rook2.isValidMove(4, 7))
11                  rook2.move(4,7);
12              System.out.println("rook2 location: " + rook2);
13              System.out.println();
14
15              Piece rook3 = new Piece(4,6);
16              if (rook3.isValidMove(8, 12))
17                  rook3.move(8,12);
18              System.out.println("rook3 location: " + rook3);
19          }
20      }
```

Testing Method Inheritance

Program Output:

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

Rook class: isValidMove() method
Piece class: isValidMove() method
Piece class: move() method
rook2 location: (4,7)

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

Extension Through Overriding

Keyword

super: A reference to an object's parent class; just like `this` is a reference to itself, `super` refers to the attributes and methods of the parent.

Extension Through Overriding - A Better Design

Can we further improve our design to have better encapsulation?
Why should you require the person using your class have to explicitly check if the move is valid?
Can you incorporate this logic into your `move` method?

```
public class Piece {
    // attributes and other methods

    public boolean move(int toRow, int toColumn) {
        System.out.println("Piece class: move() method");
        if (!isValidMove(toRow, toColumn))
            return false;
        this.currentRow = toRow;
        this.currentColumn = toColumn;
        return true;
    }
    public boolean isValidMove((int toRow, int toColumn) {...}
}
```

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)
```

Pitfall: Method Overriding

```
1  public class Piece {
2      public boolean isValidMove(int currentRow, int currentColumn) {
3          <block of code to execute>
4      }
5  }
```

Overriding can't change return type:

```
1  public class Rook extends Piece {
2      public int isValidMove(int currentRow, int currentColumn) {
3          <block of code to execute>
4      }
5  }
```

Except when changing to a **subclass** of the original

Topics Covered

This Lecture

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

Next Lecture

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

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