SWEN20003
Object Oriented Software Development

Inheritance and Polymorphism

Shanika Karunasekera
karus@unimelb.edu.au

University of Melbourne

The Road So Far

- Subject Introduction
- Java Introduction
- Classes and Objects
- Arrays and Strings
- Input and Output
- Software Tools

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;
    ...



}
```

Are you a terrible programmer? No, you're just inexperienced, and have not learnt how to use Inheritance.

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.

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

- Subclass automatically contains all (public/protected) instance variables and methods in the base class
- 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

How do we Inherit Attributes?

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.

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 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 because a Piece "is not a" R
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 `Piecess`: `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          boolen 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          boolen 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          boolen 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

Recap

Previous Lecture:

- Introduction and Motivation
- Inheriting Attributes
- Inheriting and Overriding Methods

This Lecture:

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

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 overriden by subclasses.

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

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 boolean makeMove(int fromcurrentRow, int fromcurrentColumn,
5                               int tocurrentRow, int tocurrentColumn) {
6           if (board[fromcurrentRow][fromcurrentColumn] == null) {
7               return false;
8           }
9
10          Piece movingPiece = board[fromcurrentRow][fromcurrentColumn];
11
12          if (movingPiece.move(tocurrentRow, tocurrentColumn)) {
13              board[fromcurrentRow][fromcurrentColumn] = null;
14              board[tocurrentRow][tocurrentColumn] = movingPiece;
15              return true;
16          } else {
17              return false;
18          }
19  }
20
21
```

Assess Yourself

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

```
  |a|b|c|d|e|f|g|h|
  -----------------
8|R|N|B|Q|K|B|N|R|
  -----------------
7|P|P|P|P|P|P|P|P|
  -----------------
6| | | | | | | | |
  -----------------
5| | | | | | | | |
  -----------------
4| | | | | | | | |
  -----------------
3| | | | | | | | |
  -----------------
2|P|P|P|P|P|P|P|P|
  -----------------
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 TestInheritace {
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 TestInheritace {
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 is because it is defined in the Object class.

```
1  public class TestInheritace {
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  getClass() != o.getClass()) return false;
5       Piece piece = (Piece) o;
6       return (currentRow == piece.currentRow &&
7               currentColumn == piece.currentColumn);
8   }
9   @Override
10  public int hashCode() {
11      return Objects.hash(currentRow, currentColumn);
12  }
```

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
```

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