Assess Yourself

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

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

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

Assess Yourself

Class: `Chess (main)`

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

Assess Yourself

Class: `Player`

- Attributes
  - colour
- Methods
  - makeMove

Assess Yourself

Class: `Board`

- Attributes
  - pieces
- Methods
  - getNextMove
  - isGameOver

Assess Yourself

Class: `Pawn`

- Attributes
  - isAlive
  - isWhite
  - row
  - col
- Methods
  - isValidMove

Assess Yourself

Class: `Rook`

- Attributes
  - isAlive
  - isWhite
  - row
  - col
- Methods
  - isValidMove

Assess Yourself

Class: `Bishop`

- Attributes
  - isAlive
  - isWhite
  - row
  - col
- Methods
  - isValidMove

Assess Yourself

Class: `Knight`

- Attributes
  - isAlive
  - isWhite
  - row
  - col
- Methods
  - isValidMove

Assess Yourself

Class: `Queen`

- Attributes
  - isAlive
  - isWhite
  - row
  - col
- Methods
  - isValidMove

Assess Yourself

Class: `King`

- Attributes
  - isAlive
  - isWhite
  - row
  - col
- Methods
  - isValidMove

SWEN20003
Object Oriented Software Development

Inheritance

Semester 1, 2019

The Road So Far

- OOP and Java Foundations
- Classes and Objects
  - Privacy and Immutability
  - Multi-class systems

Lecture Objectives

After this lecture 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**

In-class code found here

Important

Keep the project in the back of your mind during this lecture!

Assess Yourself

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;
    ...

    private ???[][] board;

}
```

You are officially a terrible programmer. Nah you're just inexperienced, I forgive you my young padawans.

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 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

Assess Yourself

How could we use inheritance in the chess game example?

What behaviour/properties could be "generalised" across multiple classes?

Using Inheritance

Superclass

```
public class Piece {
    public int row;
    public int col;

    public boolean isValidMove(int toRow, int toCol) {
        return true; // Dummy method, the piece type isn't known
    }
}
```

Subclass

```
public class Rook extends Piece {
    public boolean isValidMove(int toRow, int toCol) {
        return (this.row == toRow) || (this.col == toCol);
    }
}
```

Using Inheritance

Keyword

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

Is A

- 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 only be **one thing**

Constructors

Do we copy and paste parent constructors into subclass constructors?

Of course not!

Keyword

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

Constructors

```
public class Piece {
    public Piece(int row, int col) {
        this.row = row;
        this.col = col;
    }
}
```

```
public class Rook extends Piece {
    public Rook(int row, int col) {
        super(row, col);
        <block of code to execute>
    }
}
```

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

Variable Declaration

Valid code:

```
Rook rook = new Rook(0, 0);
```

```
Piece piece = new Rook(0, 0); // Rook "is a" Piece
```

Invalid code:

```
Rook rook = new Piece(0, 0); // Doesn't make sense
```

Accessing Variables

```
public class Piece {
    public int row;
    public int col;
}
```

```
Rook rook = new Rook(0, 0);
System.out.format("Rook at %d,%d", rook.row, rook.col);
```

Accessing Variables

```
public class Piece {
    private int row;
    private int col;
}
```

```
Rook rook = new Rook(0, 0);
System.out.format("Rook at %d,%d", rook.row, rook.col);
```

Nope!

Accessing Variables

```
public class Piece {
    private int row;
    private int col;
}
```

```
Rook rook = new Rook(0, 0);
System.out.format("Rook at %d,%d", rook.getRow(), rook.getCol());
```

Valid, but why would we do this?

Access control!

Accessing Variables

```
public class Piece {
    private int row;
    private int col;
}
```

```
public class Rook extends Piece {
    public void moveRow(int change) {
        this.row += change;
    }
}
```

Nope!

Accessing Variables

```
public class Piece {
    private int row;
    private int col;
}
```

```
public class Rook extends Piece {
    public void moveRow(int change) {
        this.setRow(this.getRow() + change);
    }
}
```

Must use getters and setters, but why?

Accessing Variables

```
public class Piece {
    protected int row;
    protected int col;
}
```

```
public class Rook extends Piece {
    public void moveRow(int change) {
        this.row += change;
    }
}
```

All better... Right?

Accessing Variables

```
public class Piece {
    protected int row;
    protected int col;

    public setRow(int row) {
        if (row >=0 && row < BOARD_SIZE)
            this.row = row;
    }
}
```

```
public class Rook extends Piece {
    public void moveRow(int change) {
        this.row = -1000;
    }
}
```

Protected allows subclasses to subvert access control!

Assess Yourself

What level of privacy should *superclass* instance variables have?

Private!

Assess Yourself

```
public class Piece {
    private int row;
}
```

```
public class Rook extends Piece {
    private int row;
}
```

Subclass

Assess Yourself

```
public class Piece {
    private int row;

    public int getRow() {
        return this.row;
    }
}
```

```
public class Rook extends Piece {
    private int row;

    public int getRow() {
        return this.row;
    }
}
```

```
Rook rook = new Rook(0, 0);
```

Subclass

Assess Yourself

```
public class Piece {
    private int row;

    public int getRow() {
        return this.row;
    }
}
```

```
public class Rook extends Piece {
    private int row;

    public int getRow() {
        return this.row;
    }
}
```

```
Piece piece = new Rook(0, 0);
```

Subclass: the underlying object is Rook

Assess Yourself

```
public class Piece {
    private int row;

    public int getRow() {
        return this.row;
    }
}
```

```
public class Rook extends Piece {
    private int row;
}
```

```
Rook rook = new Rook(0, 0);
```

Superclass: `getRow` is called from `Piece` class!

Assess Yourself

```
public class Piece {
    private int row = 0;

    public int getRow() {
        return this.row;
    }

    public void setRow(int row) {
        this.row = row;
    }
}
```

```
public class Rook extends Piece {
    private int row = 5;

    public void print() {
        System.out.println(this.row);
    }
}
```

Assess Yourself

```
public class Program {
    public static void main(String[] args) {
        Rook rook = new Rook();

        System.out.println(rook.getRow());
        rook.print();
        rook.setRow(7);
        System.out.println(rook.getRow());
        rook.print();
    }
}
```

What does this code output?

```
0
5
7
5
```

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.

Don't. Do. It.

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

Assess Yourself

```
public class Piece {
    private int row;
}
```

```
public class Rook extends Piece {
    private int row;

    public int getRow() {
        return this.row;
    }
}
```

```
Piece piece = new Rook(0, 0);
```

Error! `Piece` doesn't have a `getRow` method!

Method Overriding

```
public class Piece {
    private int row;
    private int col;

    public boolean isValidMove(int toRow, int toCol) {
        return true; // Dummy method, the piece type isn't known
    }
}
```

```
public class Rook extends Piece {
    public boolean isValidMove(int toRow, int toCol) {
        return (this.getRow() == row) || (this.getCol() == col);
    }
}
```

Method Overriding

Keyword

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

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.

Why Overriding?

- Subclasses can **extend** functionality from a parent
- Subclasses can **override/change** functionality from a parent
- Makes the *subclass* behaviour **available** when using variables of a *superclass*
- Defines a *general* "interface" in a superclass, with *specific* behaviour implemented in the subclass

Extension Through Overriding

A **better** design:

```
public class Piece {
    public boolean isValidMove(int row, int col) {
        return row >= 0 && row < BOARD_SIZE &&
                col >= 0 && col < BOARD_SIZE;
    }
}
```

```
public class Rook extends Piece {
    public boolean isValidMove(int row, int col) {
        return super.isValidMove(row, col) &&
            ((this.row == row) || (this.col == col));
    }
}
```

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.

Pitfall: Method Overriding

```
public class Piece {
    public boolean isValidMove(int row, int col) {
    }
}
```

Overriding can't change return type:

```
public class Rook extends Piece {
    public int isValidMove(int row, int col) {
        <block of code to execute>
    }
}
```

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

Pitfall: Method Overriding

```
public class Piece {
    protected boolean isValidMove(int row, int col) {
    }
}
```

Overriding can't make methods **more** private

```
public class Rook extends Piece {
    private boolean isValidMove(int row, int col) {
        <block of code to execute>
    }
}
```

But **less** private is okay

```
public class Rook extends Piece {
    public boolean isValidMove(int row, int col) {
        <block of code to execute>
    }
}
```

Restricting Inheritance

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

Keyword

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

Restricting Inheritance

Keyword

final: Final methods may not be overriden by subclasses.

```
<privacy> final <returnType> <methodName>(<arguments>) {
    <block of code to execute>
}
```

Restricting Inheritance

Keyword

final: Final classes may not be inherited.

```
<privacy> final <ClassName> {
    <instance variables>

    <method definitions>
}
```

Assess Yourself

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

Assess Yourself

```
public class Board {
    private Piece[][] board;

    public boolean makeMove(int fromRow, int fromCol, int toRow, int toCol) {
        if (board[fromRow][fromCol] == null) {
            return false;
        }

        Piece movingPiece = board[fromRow][fromCol];

        if (isValidMove(movingPiece, toRow, toCol)) {
            board[fromRow][fromCol] = null;
            board[toRow][toCol] = movingPiece;
            movingPiece.setLocation(toRow, toCol);
        }
    }

    public boolean isValidMove(Piece piece, int toRow, int toCol) {
        return piece.isValidMove(toRow, toCol);
    }
}
```

Metrics

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

```
 |a|b|c|d|e|f|g|h|
------------------
1|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|
```

Metrics

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?