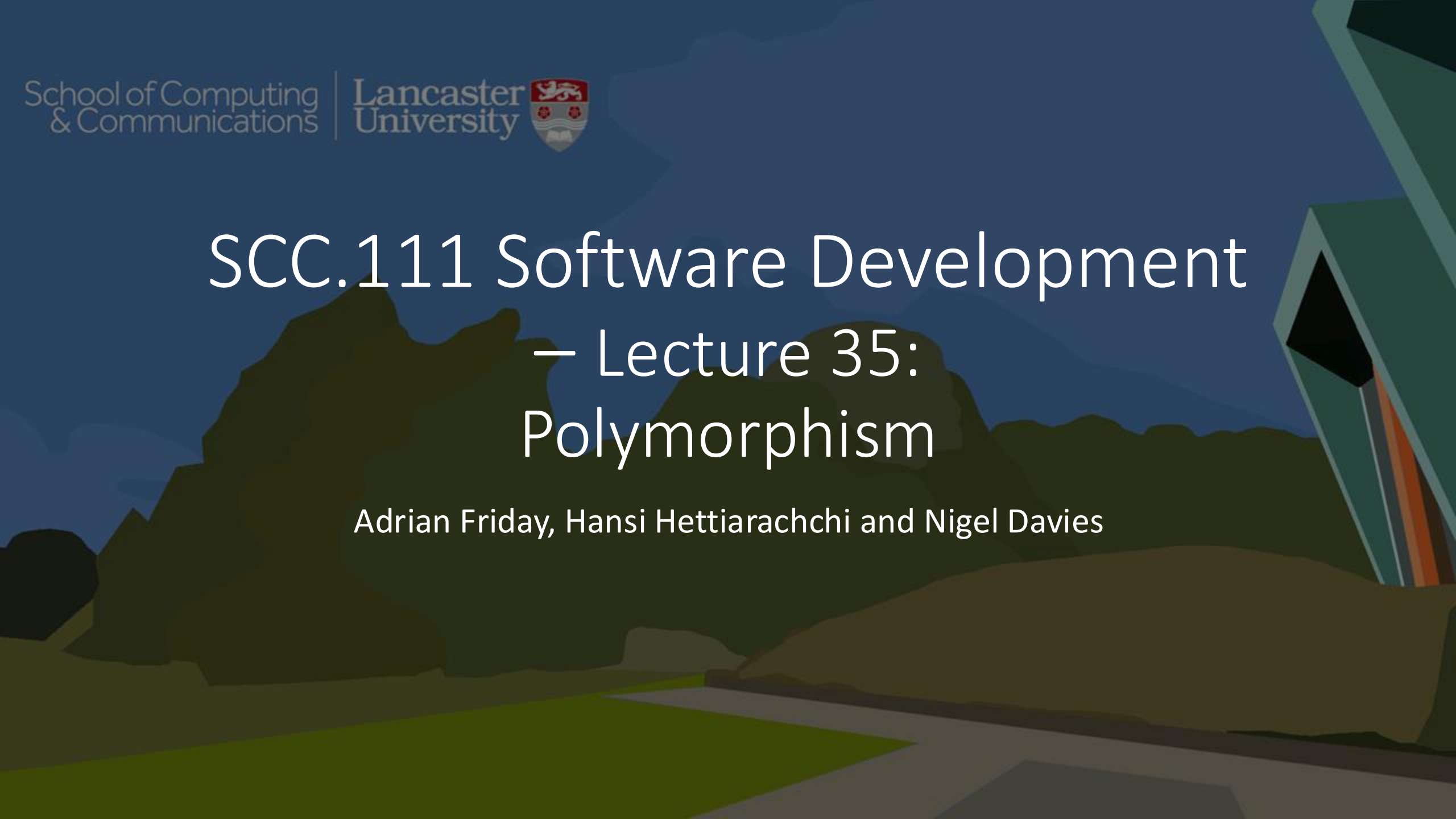



# SCC.111 Software Development – Lecture 35: Polymorphism

Adrian Friday, Hansi Hettiarachchi and Nigel Davies

# Introduction

Lancaster University

- In the last lectures, we:
  - Introduced a new concept in object-oriented programming: **inheritance**
  - Saw some examples of inheritance in practice

- Today we're going to discuss a different but heavily related concept:
  - **Polymorphism**

- We'll also reinforce some other concepts we've seen but not explored fully:
  - **Constructor chaining**
  - **Method overloading**
  - **Method overriding**

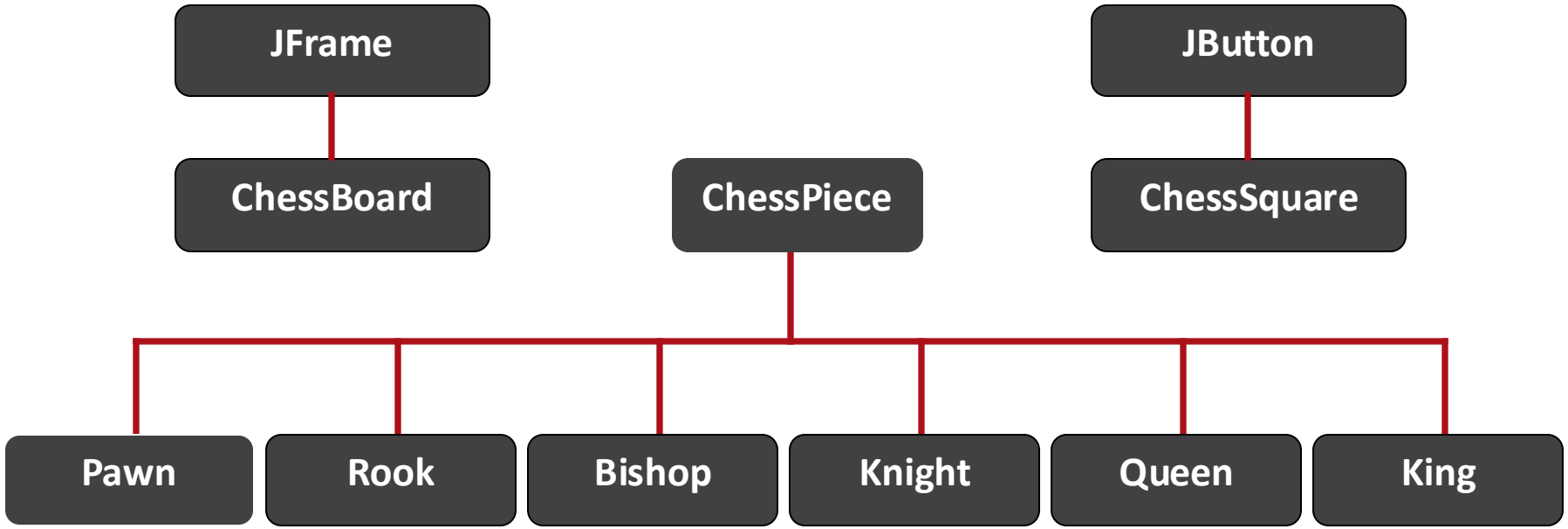
# A quick refresher…

Lancaster University

**Wizard Chess: A concrete example**

- Visually model the correct behaviour of chess pieces in Swing
- Prevent illegal moves on the board.
- **What classes can we identify?**
- **What names are sensible?**
- **Is there an inheritance relationship between any of these classes?**

# Wizard Chess: Inheritance Hierarchy…



# Constructor Chaining

Lancaster University

**Constructors provide a way to initialize objects when they are created…**

- But if we extend a class, what happens to its constructor?
- Do we write a new one for the subclass?
- Do we use the one in the superclass?

**BOTH!**

# The super keyword…

Lancaster University

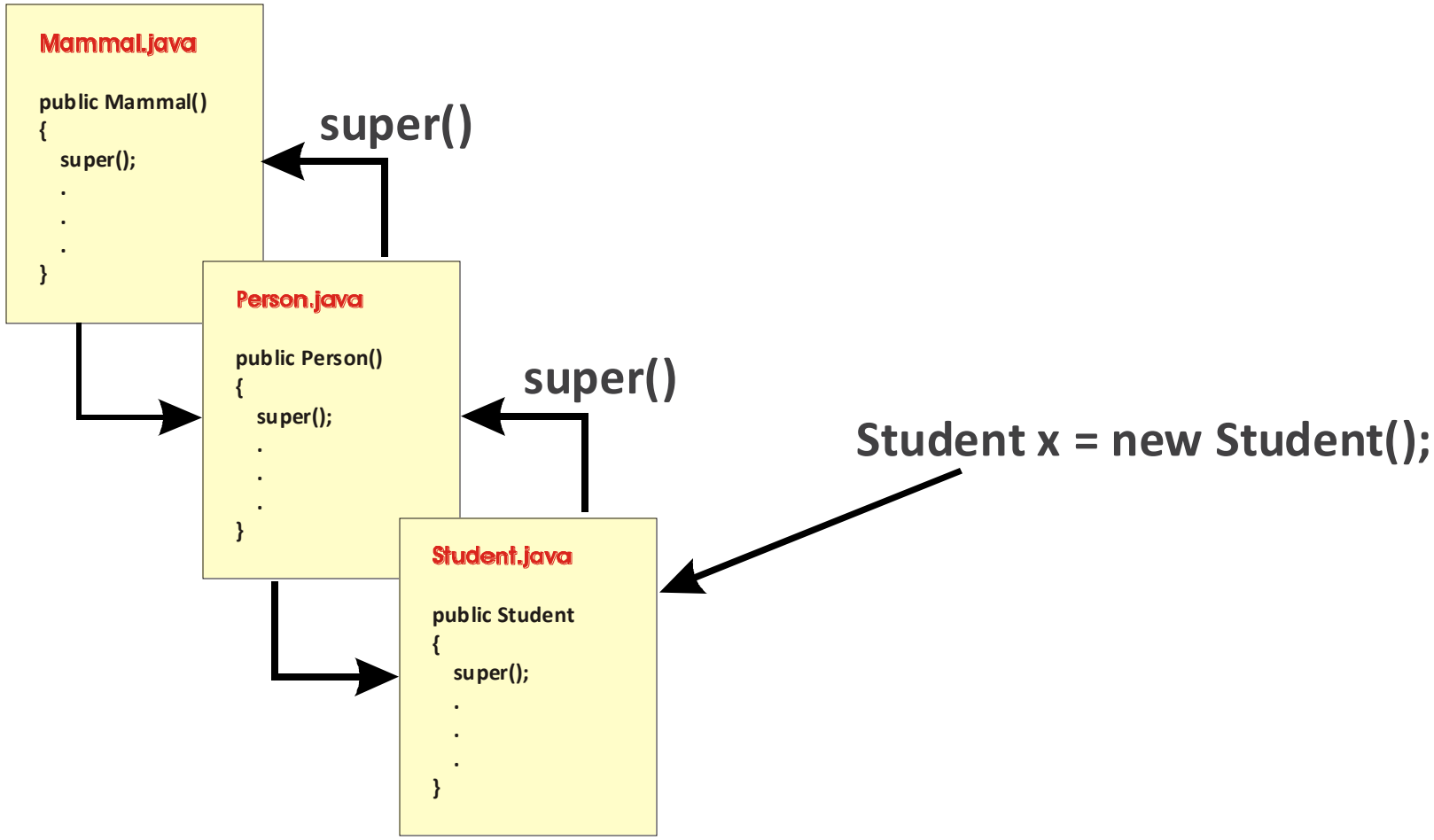
**In java, we use super() to call the constructor method of our super class**

- We can use super() to call the constructor in our superclass before initialising our own internal attributes… This guarantees super classes are initialised properly.

- If you choose to use it, super() **must** be the first statement in your constructor. Put any necessary parameters to the constructor in the brackets, just as you would using new()…

- But if you omit it, then the java compiler will put it in anyway!
  - It will assume a **default constructor** (a constructor with no parameters)

# Constructor Chaining Example



# The C++ Equivalent

Lancaster University

**In C++, we use initializer lists to do the same**

- Use the name of the super class, along with its constructor parameters when we implement the constructor:

```
Lecturer::Lecturer(string name, int age, string subject) : Person(name, age)
{
    specialistSubject = subject;
}
```

- **Think**: why does C++ need the name of the super class?

# Analysis: Consider this code extract…

Lancaster University

```
public class ChessBoard extends JFrame implements ActionListener
{
    private ChessPiece[] pieces = new ChessPiece[16];

    public ChessBoard()
    {
        // Add the p(r)awns
        for (int x = 0; x<8; x++){
            pieces[x] = new Prawn(squares[x][6]);
        }
    }
}
```

**This looks like a type mismatch…**

- pieces is an array of type **ChessPiece**
- Yet, pieces[0] is being assigned to an object of type **Prawn**
- **HOW?**

# Polymorphism

Lancaster University

**From the Greek and Latin:** **To take many forms.**

- Any subclass must have all the attributes and methods of its superclass, by definition.
- Also, a subclass can only add functionality to a class… never remove it.
- Therefore, all classes **must** have all the external behaviour of their super class.

- **Therefore, any class can be treated as a type of its super class**.

- All instances of a subclass can therefore be:
  - passed as a parameter into methods expecting its super class
  - stored in a variable or array that is typed as its superclass

- **This provides full backward compatibility of a subclass with any code written for its superclass.**

# Polymorphism: Application

Lancaster University

**Any object instance of a class that extends another can be treated as an object of that class**

- An object reference of one type can therefore refer to an object of a different type!
- This is safe, as any subclass will have at least the same methods and instance variables.

- Recall casting from last term: the process of converting a primitive variable from one type to another:
- Casting can also be used convert between types of object references…
- **but only if the underlying object is the same class or a subclass of the type you are casting to**

```
float f = 42.42;
int i = (int) f;
```

```
ChessPiece c, c2;
Prawn p, p2;

p = new Prawn();
c = p;
p2 = (Prawn) c;
```

# Polymorphism: Application…

Lancaster University

**Remember we have an inheritance hierarchy?**

- Java will **implicitly** and **transparently** cast an object reference as necessary, provided that the transformation goes upwards the inheritance hierarchy...

- Casting down the hierarchy must be done **explicitly**. This is also inherently dangerous… **why?**
- Casting of object references cannot be undertaken outside a polymorphic relationship
  - For example, a Prawn **cannot** be cast to a Queen

- Note: It is only the **object reference** which changes type during casting.
  - **Once an object instance is created, it lives and dies with the same class and cannot be changed!**
  - Polymorphism can therefore be viewed as an ‘overlay’ that only looks at part of an object’s implementation.

# Polymorphism: Example…

Lancaster University

```
public class ChessBoard extends JFrame implements ActionListener
{
    private ChessSquare[][] squares = new ChessSquare[8][8];
    private ChessPiece[] pieces = new ChessPiece[16];
    private ChessPiece pieceMoving = null;

    public void actionPerformed(ActionEvent e)
    {
        ChessSquare b = (ChessSquare)e.getSource();

        if(pieceMoving == null)
        {
            for (int i=0; i<pieces.length; i++){
                if (pieces[i].square == b)
                    pieceMoving = pieces[i];
            }
            return;
        }

        if (pieceMoving.canMoveTo(b))
        {
            pieceMoving.moveTo(b);
            pieceMoving = null;
        }
    }
}
```

# Method Overriding

Lancaster University

**Inheritance allows us to specialize the behaviour of an existing class by adding stuff**

- We can add new instance variables and methods in any subclass we create.
- It's not possible to remove methods and instance variables. You can choose to not use them, but they always implicitly remain.

- **Method overriding** can however be used to **change** behaviour by **replacing** a method in the superclass.

- Implementation is simple: **define a method in your class with the same signature as one in a superclass**
- The functionality in this method replaces the functionality inherited from the superclass.

# Method Overriding Example

Lancaster University

```
public class ChessPiece
{
    public boolean canMoveTo(ChessSquare s)
    {
        return false;
    }
}
```

```
public class Prawn extends ChessPiece
{
    public boolean canMoveTo(ChessSquare s)
    {
        if(square.xLocation == s.xLocation &&
          (square.yLocation == s.yLocation + 1 || (square.yLocation == 6 && s.yLocation == 4)))
            return true;

        return false;
    }
}
```

# Polymorphism and Method Overriding

Lancaster University

**If we have an object reference of type ChessPiece that is referring to an object of type Prawn…**

- And we call the canMoveTo() method using that reference…
- Which method is executed? The one defined in ChessPiece or the one defined in Prawn?

**The underlying object instance never changes type**

- The method called will be the most specific (lowest in the inheritance hierarchy) applicable to that object instance.
- **Exactly the same** functionality as if the object reference were the same type as the object instance.
- However, only access to the methods and instance variables matching the object reference's position in the inheritance hierarchy will be accessible.

- **This enables us to create heterogeneous collections of object instances.**
  - **Objects in the  aren't necessary all of the same type, but enough functionality to allow them to be interoperable…**

# Method Overloading…

Lancaster University

**Not to be confused with Method Overriding!**

- Java methods are uniquely identified by their **class**, **name** and **parameter list**
- The name and parameter list is often referred to as a method's signature.
- This means we can have more than one method with the same name, in the same class as long as they have different parameter lists.

- **Defining multiple methods with the same name but different parameter lists is known as method overloading.**
  - method overloading is often applied to constructors as we've already seen
  - but can be applied to any method

# Summary

Lancaster University

**Today we learned about:**

- Principles of **polymorphism**
- How it can provide future-proofed extensibility in combination with inheritance
- How method overriding can allow us to tailor the implementation of existing classes

# Wizard Chess: Methods

Lancaster University

**Which class do you think these methods should be defined in and why?**

**Attributes:**

- The chess square a piece is residing on. — **ChessPiece**
- A two dimensional array of 64 chess squares. — **ChessBoard**
- Image representing a piece. — **ChessPiece**
- The [x,y] location of a chess square — **ChessSquare**

**Methods:**

- ActionListener() — **ChessBoard**
- moveTo() — **ChessPiece**
- canMoveTo() — **All ChessPiece subclasses!**