SCC.111 Software Development

— Lecture 36: Interfaces

Adrian Friday, Hansi Hettiarachchi and Nigel Davies

Introduction

- In the last lectures, we:
  - Introduced a new concept in object-oriented programming: **polymorphism**
  - We also discussed method overriding and method overloading.

- Today we'll complete the holy trinity of OO concepts by discussing:
  - **Abstract classes**
  - **Abstract methods**
  - **Interfaces**

SCC111 Quiz Assessment

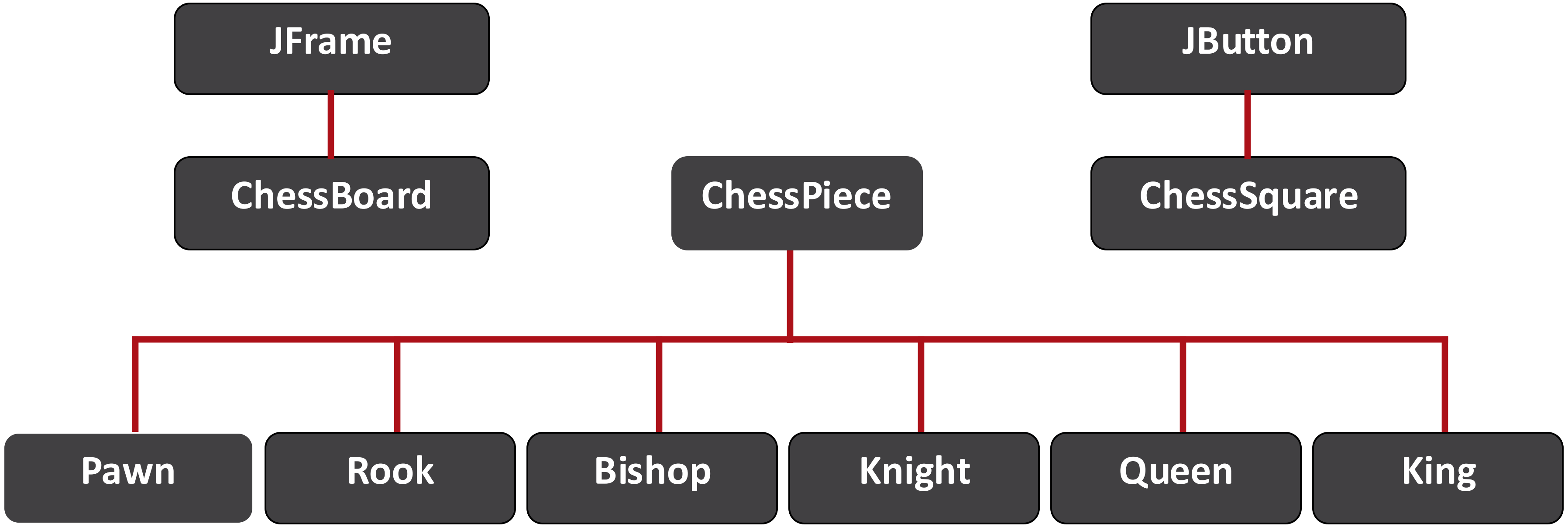
But first a few words about this terms remaining SCC111 assessment

- There will be an in-lab quiz assessment in Week 20.
- Combination of knowledge/analysis and problem solving as in week 15.

- **The problem-solving component will be based on some software development work you should undertake in the week before the quiz (Week 19).**

- We will be announcing this as the Week 19 lab task on Monday.
- It will take the form of some (moderately complex) Object Oriented program in the Java programming language.
- Well will ask you to submit and augment this program during your lab session.

Wizard Chess: One more time! :D



Abstract Classes

Consider: Will we ever actually make an instance of the ChessPiece class? Or just its subclasses?

- There is no good reason to…and allowing a programmer to do so would likely just confuse them, and encourage them to use your classes in ways that your code might not expect
- The ChessPiece class was specifically **designed to always be extended** – it has no reason to exist otherwise.
- We call such classes **abstract classes**.

- Abstract classes *can only* be extended. **It is not possible to create objects of this type using new**.
- if you try, you'll get a compile time error.
- Java (and C++) use the **abstract** keyword to define classes as abstract:

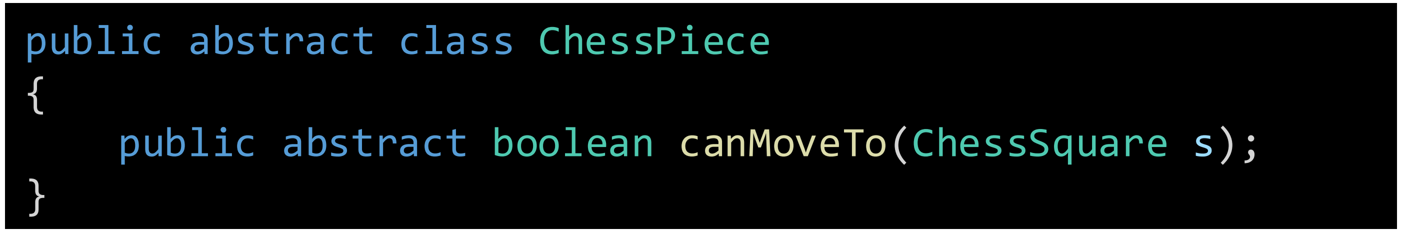
```
public abstract class ChessPiece
{

}
```

Abstract Methods

Similarly, we can define methods which can never be called!

- These are called **abstract methods**.
- Any subclasses extending a class containing an abstract method **must** override that method.
- This is enforced at compile time.

- Abstract methods provide a placeholder only – guaranteeing that any subclass has an implementation of that method…
- This is particularly useful for methods that you know are needed, but do not make sense to implement at a high level of abstraction.



```java
public abstract class ChessPiece
{
    public abstract boolean canMoveTo(ChessSquare s);
}
```

Limits of Inheritance Based Polymorphism

Remember, Java is a single inheritance language…

- Classes can extend precisely one other class. No more.
- So what if we want to build a class capable of many roles?

Separating Polymorphism from Inheritance

- Inheritance guarantees that a subclass has all the external capabilities (methods and attributes) of a superclass.
- This means polymorphism can occur without compromising type safety.
- Need inheritance be the **only** way to guarantee a class has specific capabilities?

Interfaces: Definition

An Interface is a named specification of zero or more methods

- Interfaces are similar to abstract classes, but do not contain any functionality.
- Interfaces **do not** contain any attributes
- Interfaces **do not** contain any method **implementations**[1], they only contains methods **description** that classes implement.

- A class can declare that is wishes to implement any number of interfaces.
- By contract, that class **must** then implement the methods specified in those interfaces. This is enforced by the compiler.

- **We can therefore guarantee that any class implementing an interface has all the methods defined in that interface.**

[1] normally!

Interfaces and Polymorphism

Polymorphism in Java is also applied to interfaces.

- As we can definitively say a class has the capabilities defined in an interface, we can apply polymorphism without compromising type safety.
- **Therefore, a class can be treated as a type of any interface it chooses to implement.**
- As classes can implement many interfaces, this provides a clean mechanism that addresses the limitations of single inheritance languages, whilst sidestepping the complexity of multiple inheritance.

A complete definition of Polymorphism in Java:

- **A class can be treated as a type of its class, any of its super classes, or any of the interfaces it implements.**

Interfaces: Example

The Java API contains a multitude of interfaces…

- **ActionListener** is, in fact, an interface.
- This interface specifies a single method only:

```
interface ActionListener
{
    public void actionPerformed(ActionEvent e);
}
```

- Classes that implement the ActionListener interface must therefore write an actionPerformed method.
- **In return, objects of that class can be treated as a type of ActionListener.**

Interfaces: Example…

The Java API contains a multitude of interfaces…

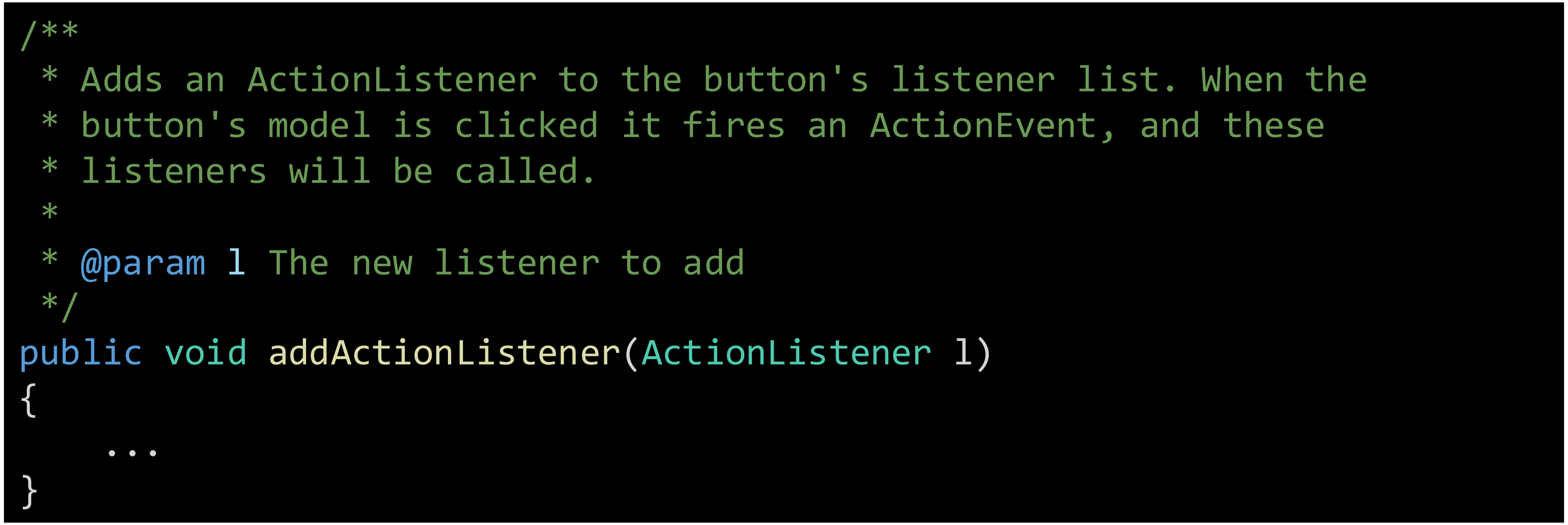
- Classes that implement the ActionListener interface **must** therefore write an **actionPerformed** method.
- In return, that class can be treated as a type of ActionListener (by polymorphism):

```
public class ChessBoard implements ActionListener
{
    ChessSquare[][] squares = new ChessSquare[8][8];

    public ChessBoard()
    {
        squares[x][y] = new ChessSquare(x, y, "pieces/EmptySquare.jpg");
        squares[x][y].addActionListener(this);
    }

    public void actionPerformed(ActionEvent e)
    { … }
}
```

Interfaces: Example….

Consider this extract from the AbstractButton source code:

- **QUESTION**: What do you think the super class of JButton might be?
- **QUESTION**: What do you think the implementation would conceptually look like?



```
/**
 * Adds an ActionListener to the button's listener list. When the
 * button's model is clicked it fires an ActionEvent, and these
 * listeners will be called.
 *
 * @param l The new listener to add
 */
public void addActionListener(ActionListener l)
{
    ...
}
```

Interfaces: Example…..

Consider what is happening here.

- The addActionListener method (in AbstractButton) takes a single parameter – an object reference to an object that implements the **ActionListener** interface.

- It can then store these (in some sort of array/collection of type **ActionListener**), until needed.
- It can then invoke the **actionPerformed** method on those instances when the time is right (in this case, when the button is clicked).

- The AbstractButton class has no knowledge of our class (ChessBoard) when it was written in (2002!)
- Yet it can call methods on our class in a **type safe** manner.

Roll Your Own Interfaces

It is very straightforward to create your own interfaces

- Interfaces can be used anytime to promote polymorphism but avoid restrictions of single inheritance.
- **To promote extensibility and elegance.**

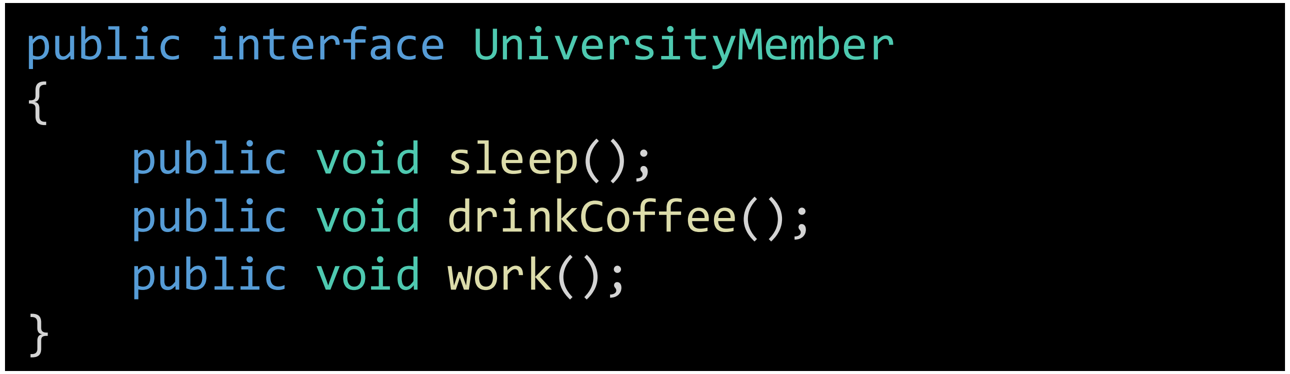
- Interfaces allow you to define what other programmers must do in order to interface with **your** code. Then your code can support working with objects that haven’t even been written yet. And people will like you.

- **Remember:**
- Interfaces define only methods. No instance variables allowed
- Interfaces are merely a specification. No implementation of the methods.
- If you need to provide default behaviour, it's usually better to use inheritance instead.

Custom Interface Example

Interfaces are defined much like classes

- Interfaces are named, in capitalized camel case.
- Filename should match interface name.
- List the method signatures that you want to be part of that interface.



```java
public interface UniversityMember
{
    public void sleep();
    public void drinkCoffee();
    public void work();
}
```

Custom Interface Example…

Lancaster University

Classes use the implements keyword to declare that they want to implement your interface.

- They can then be treated as a type of your new interface.
- The compiler will enforce that the class then implements methods matching your interface.
- The specific **behaviour** of the method is still left to the programmer of the class.

```
class Undergrad implements UniversityMember
{
    int stressLevel = 0;

    public void sleep()
    {
        for (int hours=0; hours<15; hours++)
        {
        }
    }

    public void drinkCoffee()
    {
        stressLevel++;
    }

    public void work()
    {
    }
}
```

Summary

Today we learned about:

- The concept of **interfaces**
- How they allow us to define a set of expected methods, but not their behaviour.
- How polymorphism also applies to interfaces as well as inheritance

Next Lecture:

- Some more worked examples to help these ideas become more embedded