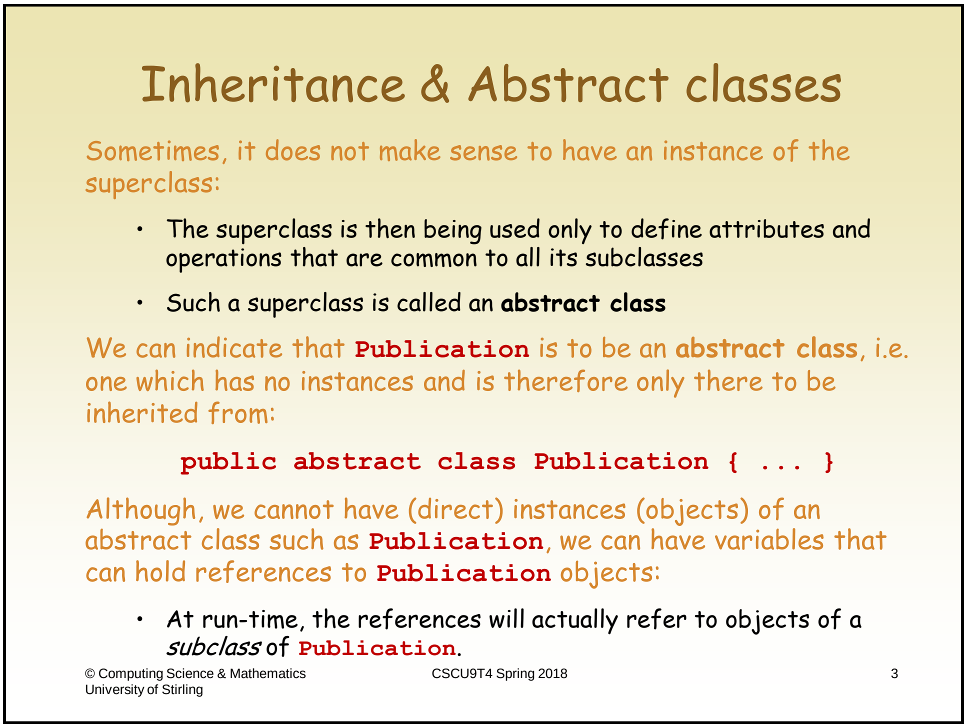
CSC9T4: Object Modelling, principles of OO design and implementation

Inheritance, Abstract, Interfaces and Multiple Inheritance



Inheritance & Abstract classes

Sometimes, it does not make sense to have an instance of the superclass:

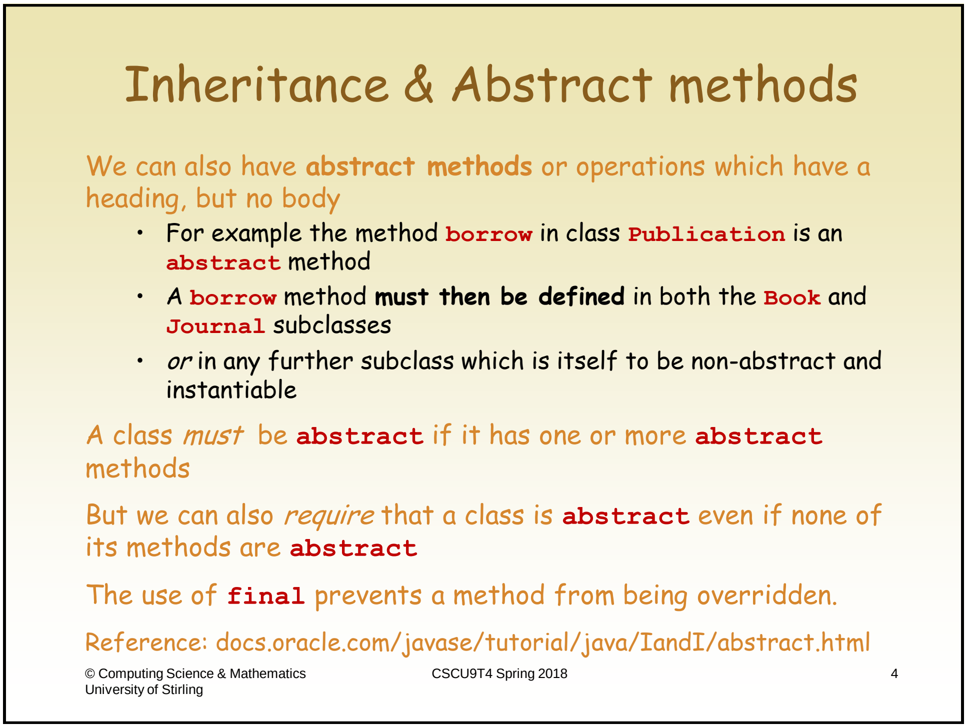
- The superclass is then being used only to define attributes and operations that are common to all its subclasses
- Such a superclass is called an **abstract class**

We can indicate that **`Publication`** is to be an **abstract class**, i.e. one which has no instances and is therefore only there to be inherited from:

```
public abstract class Publication { ... }
```

Although, we cannot have (direct) instances (objects) of an abstract class such as **`Publication`**, we can have variables that can hold references to **`Publication`** objects:

- At run-time, the references will actually refer to objects of a *subclass* of **`Publication`**.



Inheritance & Abstract methods

We can also have **abstract methods** or operations which have a heading, but no body

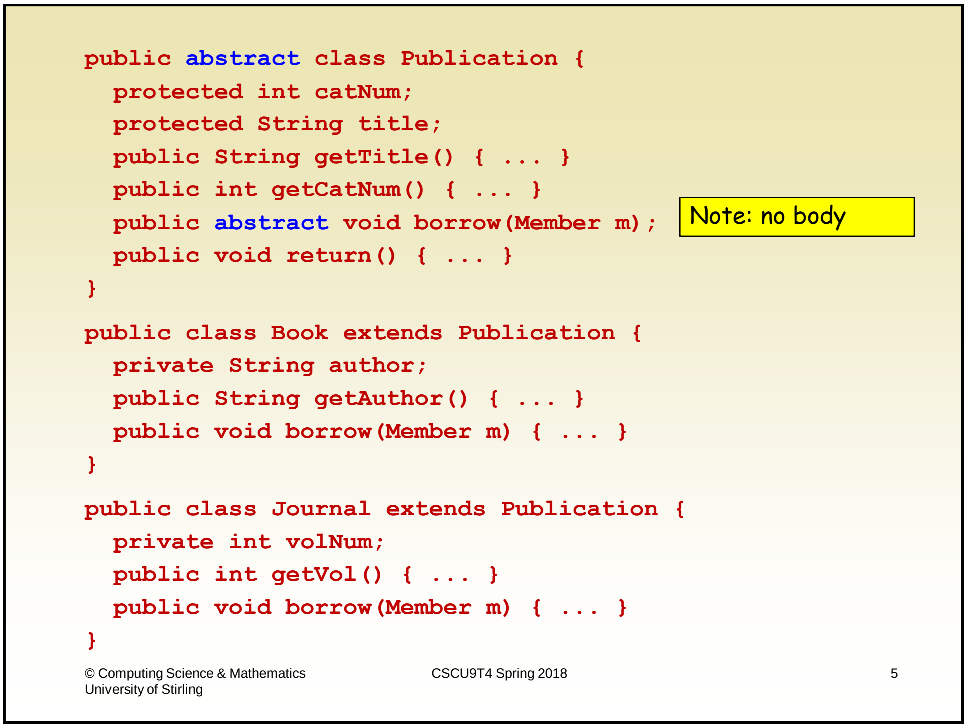
- For example the method **`borrow`** in class **`Publication`** is an **`abstract`** method
- A **`borrow`** method **must then be defined** in both the **`Book`** and **`Journal`** subclasses
- *or* in any further subclass which is itself to be non-abstract and instantiable

A class *must* be **`abstract`** if it has one or more **`abstract`** methods

But we can also *require* that a class is **`abstract`** even if none of its methods are **`abstract`**

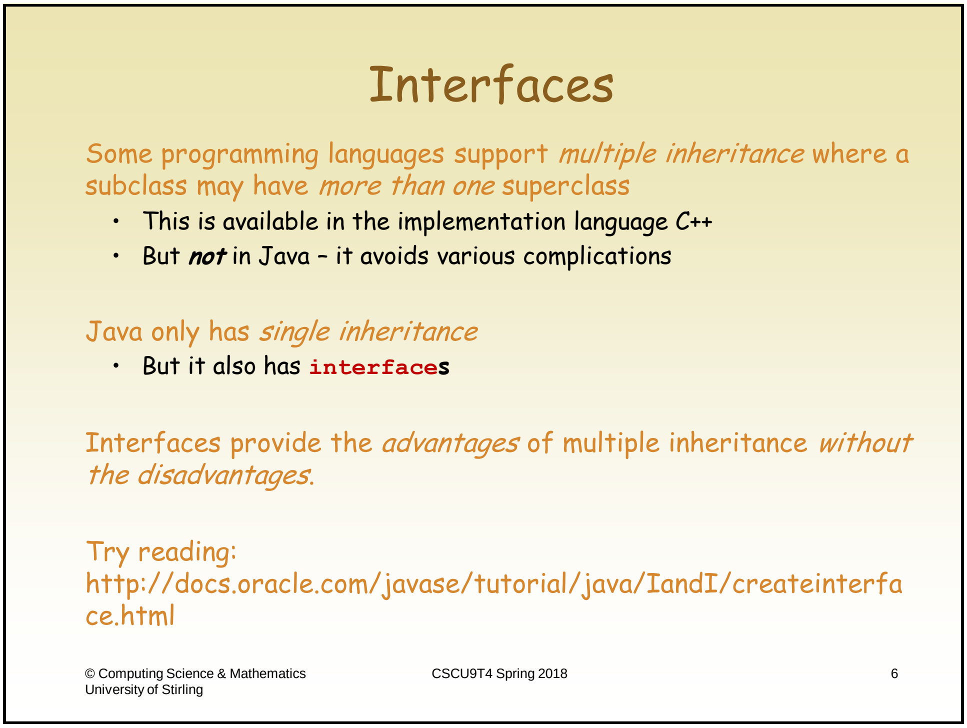
The use of **`final`** prevents a method from being overridden.

Reference: docs.oracle.com/javase/tutorial/java/IandI/abstract.html



```
public abstract class Publication {
  protected int catNum;
  protected String title;
  public String getTitle() { ... }
  public int getCatNum() { ... }
  public abstract void borrow(Member m);
  public void return() { ... }
}
public class Book extends Publication {
  private String author;
  public String getAuthor() { ... }
  public void borrow(Member m) { ... }
}
public class Journal extends Publication {
  private int volNum;
  public int getVol() { ... }
  public void borrow(Member m) { ... }
}
```

Note: no body



Interfaces

Some programming languages support *multiple inheritance* where a subclass may have *more than one* superclass

- This is available in the implementation language C++
- But ***not*** in Java – it avoids various complications

Java only has *single inheritance*

- But it also has `interface`s

Interfaces provide the *advantages* of multiple inheritance *without the disadvantages.*

Try reading: http://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html

Interfaces & Abstract Classes

An **abstract class**

- May have attributes
- And some of its operations may have implementations

An **interface** is like a class

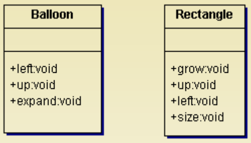
- But has **no attributes** (except `final` constants)
- And *none* of its operations have implementations

An interface is therefore like a *very abstract class*

- It simply lists the public operations that an "extending" class ***must provide implementations for***
  (but we use `implements` rather than `extends`)
- In effect it summarises a set of capabilities, a "contract"

A class that offers actual methods for the public operations "promised" by an interface is said to implement that interface

Interfaces - Example



Balloon

+left:void
+up:void
+expand:void

Rectangle

+grow:void
+up:void
+left:void
+size:void

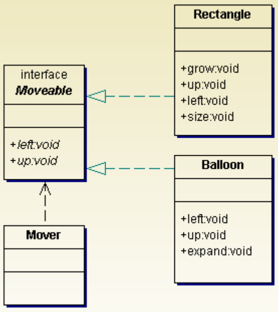
Suppose that we have classes `Rectangle` and `Balloon`.

- They have various attributes and operations
- Suppose that we have a class `Mover` whose purpose is to move objects around by calling operations `left` and `up`
- `Mover` can deal with any object that offers the operations `left` and `up` via the interface `Moveable`

We can represent an interface in a UML diagram in a similar way to a class, but there are only two partitions (no attributes) and we use the stereotype <<interface>>.

Interfaces - Example

- Here we have defined an interface called `Moveable`
- We show that class `Mover` depends on the interface `Moveable` by a dashed arrow from `Mover` to `Moveable`
- We show the relationship between `Moveable` and the classes `Rectangle` and `Balloon` by using a dashed (weaker) form of inheritance.



We say that `Rectangle` and `Balloon` **realise** the `Moveable` interface.

Interfaces - Example

An object of class `Mover` can call the operations `left` and `up` offered by a `Moveable` object.

- What do we mean by a `Moveable` object?
- The answer is an object such as `Rectangle` and `Balloon` that realises the `Moveable` interface.

By realising the `Moveable` interface, both `Rectangle` and `Balloon` must provide implementations for the operations `left` and `up`.

In Java, we say that `Rectangle` and `Balloon` **implements the interface** `Moveable`.

Interfaces - Example

What is going on might become clearer if we look at some Java:

```
public interface Moveable {
   public void left(int d);
   public void up(int d);
}
public class Rectangle implements Moveable {
   ... left ... up ... (full definitions)
   ... size ... grow ... (Rectangle specific items) ...
}
public class Balloon implements Moveable {
   ... left ... up ... (full definitions)
   ... expand ... (Balloon specific items) ...
}
```

And we could have a *very general class* **Mover** that has a reference to a **Moveable** as an attribute, and a main program that uses it:

```
class Mover {
   Moveable m;
   public Mover(Moveable m) {
     this.m = m;
   }
   private void moveIt() {
     m.left(17);
     m.up(25);
   }
```

Main program:

```
mR = new Mover(
       new Rectangle(...));
mB = new Mover(
       new Balloon(...));
mR.moveIt();
mB.moveIt();
```

m can refer to *any object that implements the* **Moveable** *interface*

- However, the only methods that can be called are those offered by **Moveable**
- So, **even** when **m** is pointing at a **Rectangle** object, it *cannot call* the **Rectangle** operations **size** and **grow**

Interfaces - Example

An interface **Moveable** specifies a contract, the classes **Rectangle** and **Balloon** guarantee to carry out the contract.

Objects of class **Mover** deal in terms of `Moveable` references, they have no need to know what kind of class has implemented the interface.

One advantage of interfaces is that we have shown that class `Mover` only depends on the `Rectangle` and `Balloon` operations that are in the `Moveable` interface.

If `Rectangle` or `Balloon` were modified so that one of their other operations changed, we have a guarantee that `Mover` would not be affected.

Why interfaces are useful

Interfaces are useful as they allow our designs/programs to be more general/flexible than they otherwise would be

Natural choices for interface names are ***adjectives***

For example:

- We can use an interface name, say **Moveable**, as the *type of a formal parameter*:

```
private void myMethod(Moveable m, ...) { ... }
```

- This indicates that **myMethod** is happy to receive *any object at all* as an actual parameter *provided* that it offers the operations specified in **Moveable**
- The actual parameter can be an instance of *any class* that "**implements Moveable**" (and the Java compiler checks for us!)

Interfaces & Roles

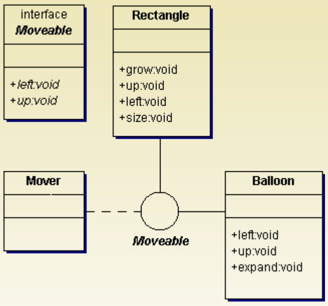
A use of interfaces is that they allow a class to play different roles.

- An object of a `Person` class may play the role of Employee.
- The operations used in that role may be defined in an `Employable` interface.
- An `Employer` object then deals with `Person` objects through their `Employable` interface rather than directly manipulating `Person` objects.
- As well as the operations shown in the `Employable` interface, `Person` objects may have operations they use when interacting with their children.
- These operations could be defined in a `Responsible` interface.

Multiple Interfaces

Many classes can realise a given interface and a given class can realise several alternative interfaces.

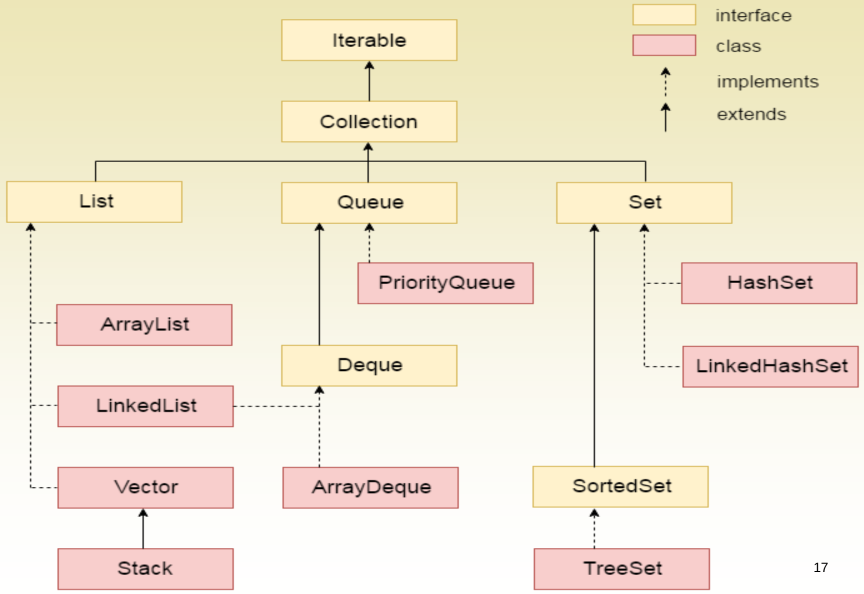
So that the UML diagram does not become too cluttered, we can represent interfaces in the following way:



The interface is collapsed into a "lollipop'' showing that `Rectangle` and `Balloon` implement the interface `Moveable`.

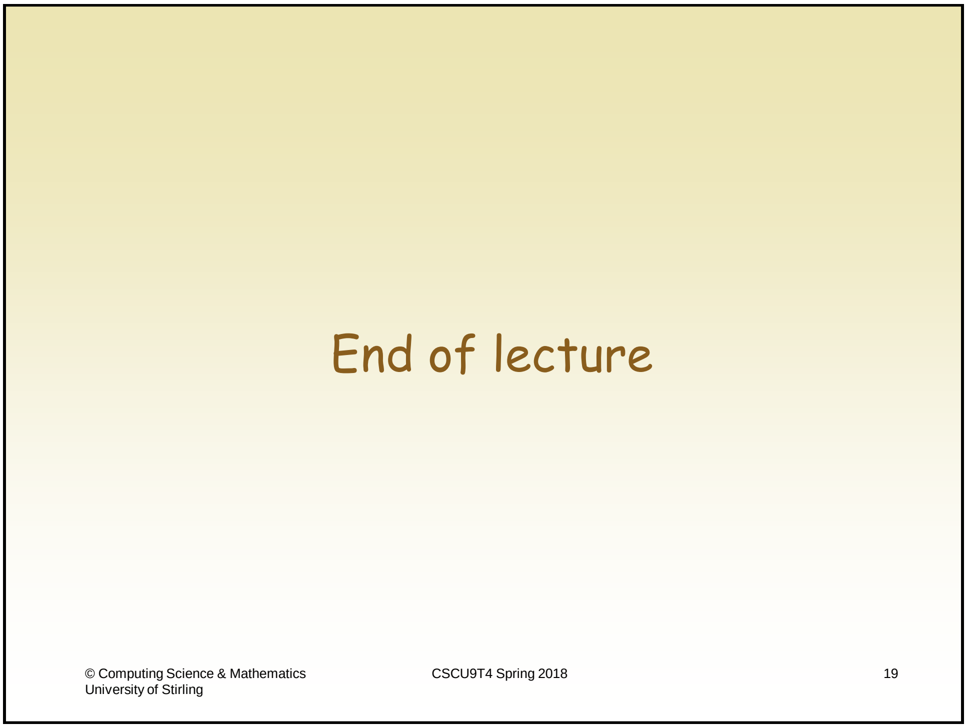
The dashed arrow shows that `Mover` depends on the interface `Moveable`.

The Collection Framework



Advantages of OOP

1. Software reuse is enhanced.
2. Software maintenance cost can be reduced.
3. Data access is restricted providing better data security.
4. Software is easily developed for complex problems.
5. Software may be developed meeting the requirements on time, on the estimated budget.
6. Software has improved performance.
7. Software quality is improved.
8. Class hierarchies are helpful in the design process allowing increased extensibility.
9. Modularity is achieved.
10. Data abstraction is possible.



End of lecture

© Computing Science & Mathematics
University of Stirling