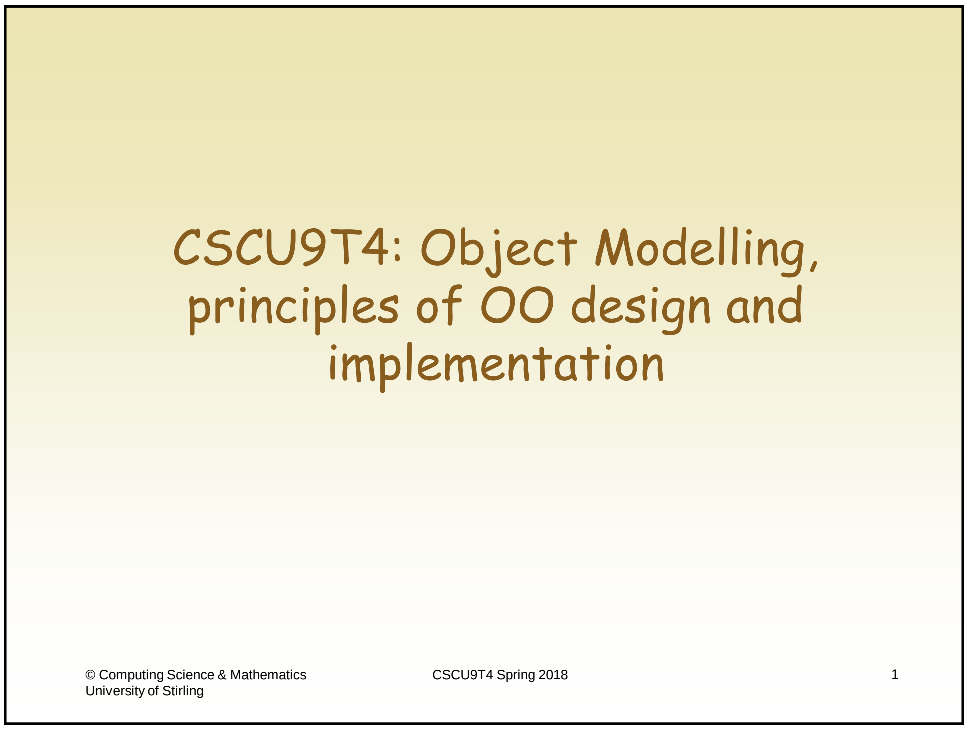
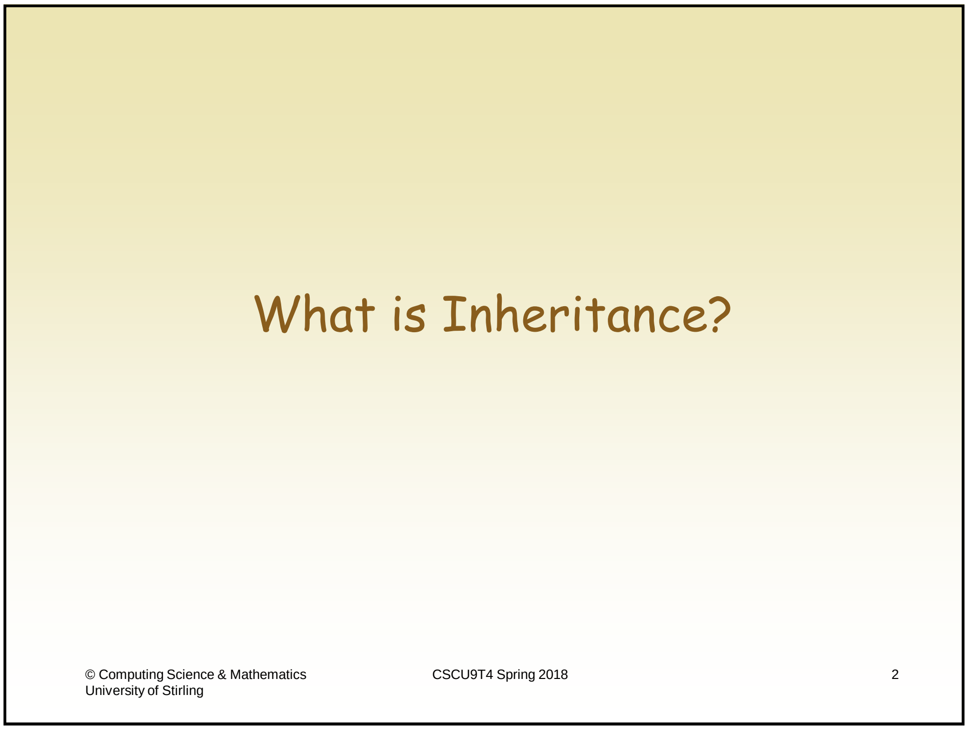


CSCU9T4: Object Modelling, principles of OO design and implementation



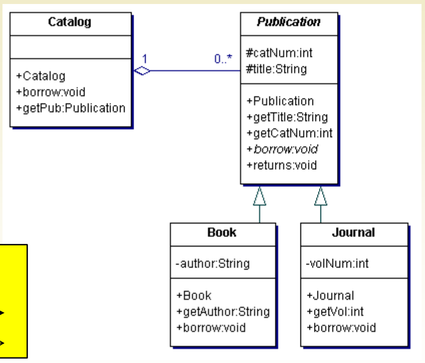
What is Inheritance?

Inheritance

In inheritance, a ***subclass*** "extends" the definition of its ***superclass***

- It implicitly includes the attributes and operations from the superclass, and may add attributes and operations. It may also redefine the implementations of operations

For example, inheritance is shown in the following, where classes `Book` and `Journal` "inherit from" `Publication`:



Note: UML notation.
Arrowheads can be
closed: inheritance
open: reference (via an attribute)

Inheritance - extends

```
public class Publication{ ... }
public class Book extends Publication { ... }
public class Journal extends Publication { ... }
```

We can have an inheritance ***hierarchy***

- If, say, `Publication` is itself a subclass of an even higher-level class
- Or, say, `Book` has subclasses

```
public abstract class Publication {
  protected int catNum;
  protected String title;
  // operations
}
public class Book extends Publication {
  private String author;
 // operations
}
public class Journal extends Publication {
  private int volNum;
 // operations
}
```

Inheritance & Attributes

Although we want to hide the attributes of a superclass from ordinary clients, we usually want to make them *visible in the subclass*

- This is *not automatic* with `private` attributes

Hence, there are *three levels of visibility*: public, private and protected.

- `public` : visible to clients
- `private` : visible only within the class
- `protected` : visible only within the class *and its subclasses*

So that the method bodies in `Book` and `Journal` can refer to `title` and `catNum`:

```
protected int catNum;
protected String title;
```

Inheritance & Operations

```
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) { ... }
}
```

Inheritance & Operations

Note that the operation **`borrow`** is given in *all three classes* **`Publication`**, **`Book`** and **`Journal`**

- That shows that it is defined in **`Publication`** ...
- And the definition is **overridden** in the **`Book`** and **`Journal`** subclasses

This is useful if we wish to indicate:

- That all **`Publications`**, whether they are a book or a journal, have a public **`borrow`** operation
- But that the *actual details* for borrowing a **`Book`** and borrowing a **`Journal`** are *different* – so the two subclasses must have separately defined methods

We also need to decide whether we need an actual *implementation* (method body) of **`borrow`** in **`Publication`**

- There does *not need to be*, and in this example there is not – indicated by the keyword **`abstract`**
- (See discussion of "abstract methods" later on)

Instantiation of subclasses

To be completely clear:

When a *subclass* is instantiated using **`new`**:

- Enough storage is allocated for *all the attributes* of the subclass and its one (or more) superclasses
- Effectively, all the subclass's operations are included, ***plus*** all the ***non-overridden operations*** from the superclasses **except** the constructors
- A subclass constructor may call its superclass's constructor using

  `super(…);`

  » (This is Java; details may vary slightly for other OO languages)

Therefore:

- An instance of a subclass *has all the properties expected of an instance of a superclass*
- For example: a **`Book`** has **`title`**, **`getTitle`**, **`catNum`**, **`return`**, **`borrow`**, etc

A consequence of the previous slide, and a feature of inheritance in object-oriented systems is that:

- Anywhere **a reference to a superclass** object is expected, we can use a reference to a **subclass** object instead
- … because we can guarantee that it has all the capabilities

Hence, a reference to a `Publication` object could refer to a `Book` object

Consider some examples: are these valid? What do they do?

```
Publication pub = new Publication("Smith");

Book pub = new Book("Smith");

Publication pub = new Book(”Smith");

Book pub = new Publication("Smith");
```

Further:

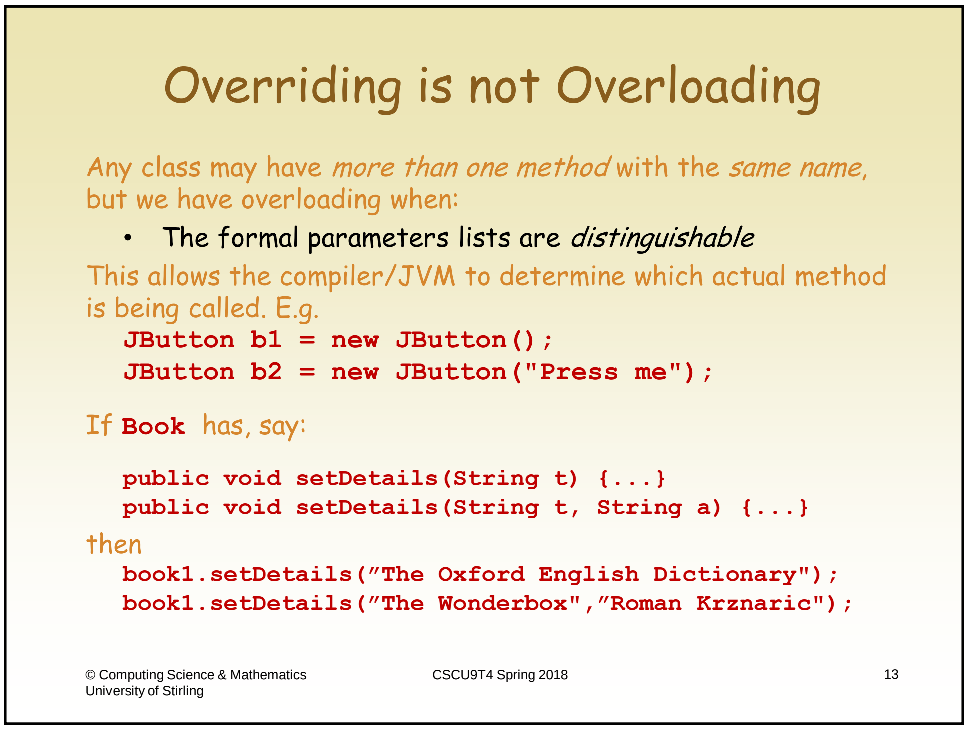
- When we call an operation through a reference, the *actual method used* depends on the *actual object referred* to

(Again, this is Java, and details may vary in other OO languages)

For example, in Java, after the declaration:

```
Publication pub = new Book(”Smith");
```

- What happens if we call the `borrow` operation:
  `pub.borrow(...);`
  - » it is the method defined in the Book class that is used and not the one defined in Publication.
- What if we added
  `pub = new Journal(”LNCS");`
- Now what happens if we call `pub.borrow();`
  - » it would be the method defined in the Journal class that would be used



Overriding is not Overloading

Any class may have *more than one method* with the *same name*, but we have overloading when:

- The formal parameters lists are *distinguishable*

This allows the compiler/JVM to determine which actual method is being called. E.g.

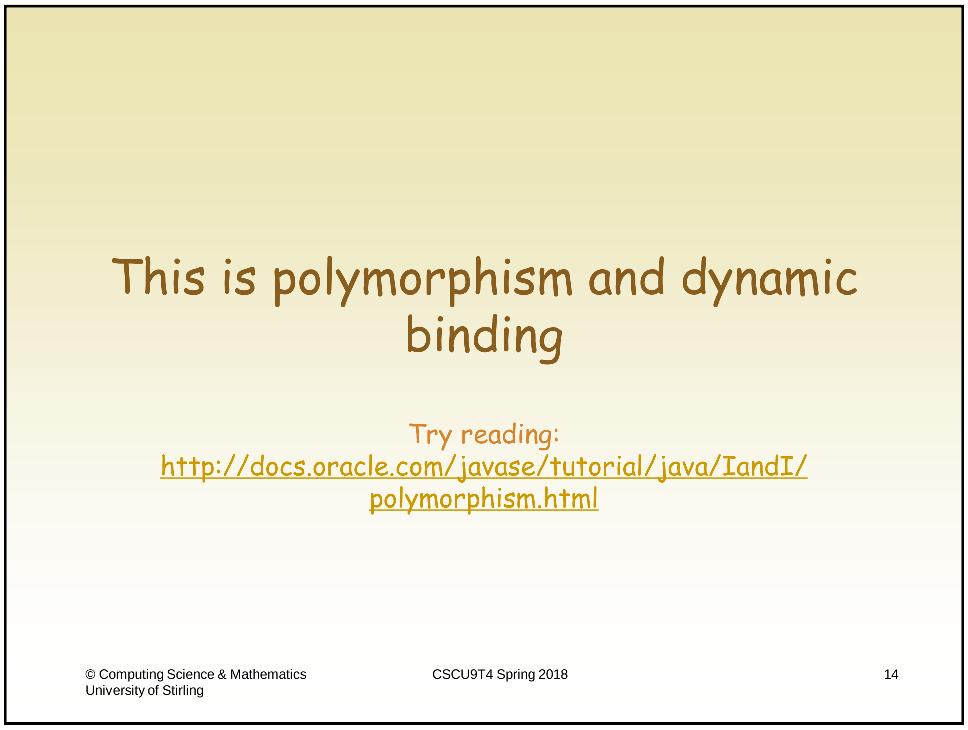
```
JButton b1 = new JButton();
JButton b2 = new JButton("Press me");
```

If **Book** has, say:

```
public void setDetails(String t) {...}
public void setDetails(String t, String a) {...}
```

then

```
book1.setDetails(”The Oxford English Dictionary");
book1.setDetails(”The Wonderbox",”Roman Krznaric");
```



This is polymorphism and dynamic binding

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

Why use Inheritance?

Discussion

End of lecture