UML 3
Inheritance, Dependencies Associations & Interfaces,

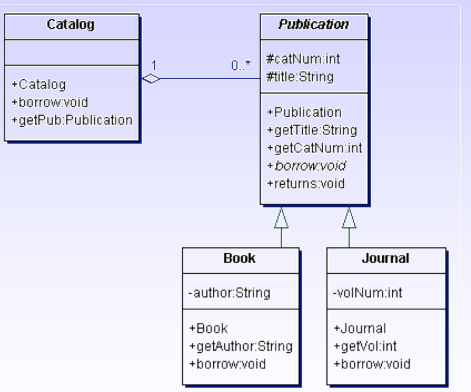
Associations - Inheritance

Another kind of association is **Inheritance**

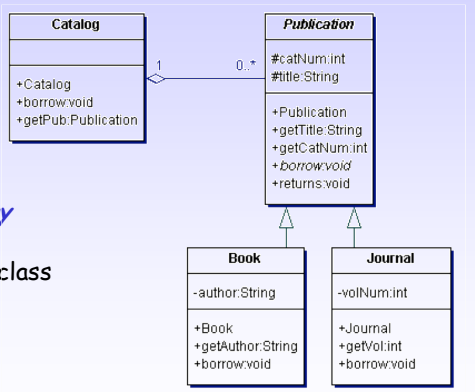
For example, inheritance is shown in the following diagram where classes Book and Journal "inherit from" Publication.

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 and may redefine the implementations of operations



Associations - Inheritance



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

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

Inheritance: visibility of attributes and operations

Note: private attributes of a superclass are *not automatically visible in subclasses*

Hence, we have *three levels of visibility*: public, private and protected.

- public: '+' in the UML
  - » visible to clients
- private '-'
  - » visible only within the class
- protected '#'
  - » visible only within the package, class *and its subclasses*

To show that the attributes title and catNum are protected, they are shown with the prefix #

Abstract classes and operations

Sometimes, it does not make sense to have an actual instance of a superclass

- In this case we designate the superclass as *abstract*
- To 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, its entry is written in *italics* in the UML class diagram (see slide 2)

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

- To indicate that the method **`borrow`** in class **`Publication`** is an abstract method, its entry is written in *italics* in the UML class diagram (see slide 2)

Catalog: Implementing an association with a superclass

Here are some aspects of an implementation of **`Catalog`**:

- There is a 1 – 0..* navigable association to **`Publication`**
  - So, **`Catalog`** could have an attribute holding an array of references
- Each actual **`Catalog`** entry can be either a **`Book`** or a **`Journal`**
- We must define the data structure to hold references to **`Publications`**, so each element can hold either a **`Book`** or a **`Journal`** (*polymorphism*):

```
private ArrayList<Publication> catalog =
                    new ArrayList<Publication>();
```

Catalog: Implementing an association with a superclass

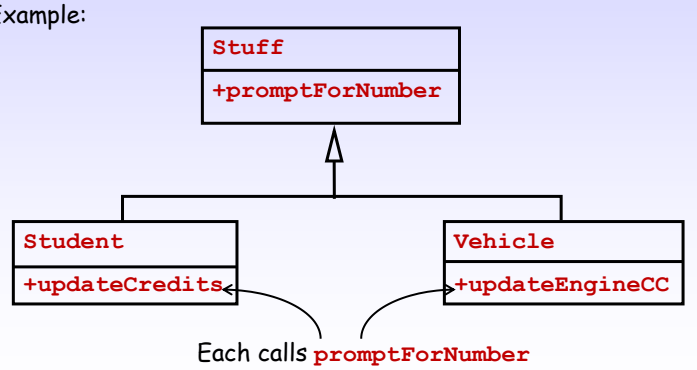
Then either **Books** or **Journals** can be added:

```
Book b = new Book(...);
catalog.add(b);
Journal j = new Journal(...);
catalog.add(j);
```


Implementation Inheritance vs …

In object-oriented *programming*, inheritance can be used *to save rewriting code*, i.e. there may be *no conceptual relationship* between a superclass and its subclasses

- This is called **implementation inheritance**.
- Example:



… Behavioural Inheritance

In this module we are *modelling* and so inheritance *must only be used* when there is *the conceptual 'is_a' relationship.*

We can say:

- **Book is_a Publication**
- A **Book** object **is_a** Publication **object**
- **Book is_a** specialisation of **Publication**
- **Publication is_a** generalisation of **Book**
- **Book is_a** subclass of **Publication**
- **Publication is_a** superclass of **Book**
- An object of class **Book** belongs to the class **Publication**

When there is an **is_a** relationship, we have **behavioural inheritance**.

In object modelling, we should **only use behavioural inheritance**.

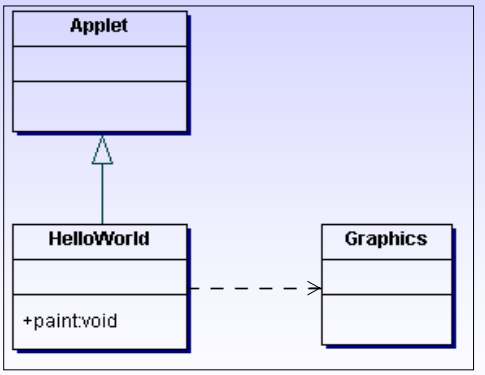
Dependency Associations
Example - Hello World

Let us take a standard simple Java applet:

```
public class HelloWorld extends Applet
{
  public void paint(Graphics g)
  {
    g.drawString("HelloWorld", 10, 10);
  }
}
```

Even this has an interesting and informative class diagram!

Dependency Associations
Example - Hello World



Dependency Associations
Example - Hello World

We show a another kind of relationship here

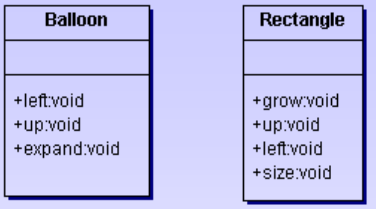
- The dashed directed line between **`HelloWorld`** and **`Graphics`** indicates a **dependency relationship**
- This means that class **`HelloWorld`** uses a variable of class **`Graphics`** *as a parameter in one of its operations (or as a local variable)*

If class **`A`** uses (depends on) class **`B`** then if **`B`** changes this may have an effect on **`A`**:

- *A* is therefore dependent on the definition/ behaviour of **`B`**
- Even if **`A`** does *not* have an attribute of class **`B`**, **`B`** could be used as a parameter or local variable ("non attribute")

In our example, changes on **`Graphics`** could have an effect on **`HelloWorld`**.

Interfaces Example



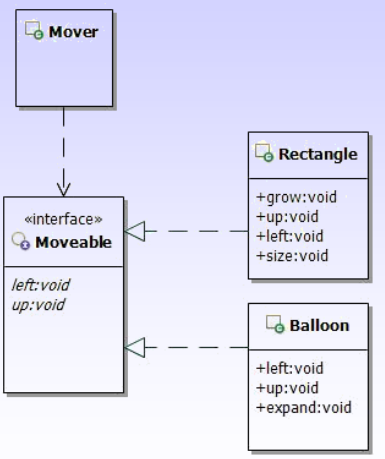
Suppose that we have classes **`Rectangle`** and **`Balloon`**

- They have various attributes and operations
- Suppose that we have a class **`Mover`** whose purpose is to move generic objects around by calling operations **`left`** and **`up`**
- So **`Mover`** can deal with *any* object that offers the operations **`left`** and **`up`**, it doesn't care what they actually are.
- To describe such objects we define an interface **`Moveable`**
- **`Mover`** then acts on **`Moveable`** objects

We can represent an interface in a UML diagram in a similar way to a class, 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 that the classes **`Rectangle`** and **`Balloon`** ***implement*** (or ***realise***) the interface **`Moveable`** by using a *dashed (weaker) form of inheritance*



Interfaces & Roles

A use of interfaces is that they allow us to classify different roles that a class may play. For example:

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

Private study: Packages

For large systems, a single class diagram can contain too many classes to be easily readable.

UML allows us to put a collection of (related) classes together into a **package**.

This helps us organise a large model.

Also, if we are working in a team, each team member can design a separate package.

Packages

A package is shown as a rectangle with a tab at the top left.

A package can contain other packages.

Packages are supported by Together

- In Java they are organised as nested folders

In Together, we usually represent the contents of each package in a separate class diagram.

We can have a top-level class diagram which consists only of packages.

- Together generates a .java file for each class.
- Together generates a folder for each package.

This corresponds exactly to what Java expects in its organisation of packages.

End of lecture