Interfaces

UML supports multiple inheritance where a subclass has more than one superclass.

This is available in the implementation language C++.

Java only has single inheritance, but like UML, it also has interfaces.

Interfaces provide the advantages of multiple inheritance without the disadvantages. Which dis/ad-\vantages?

- expressivity
- conflicts, e.g. same methods, constructors, ...

In UML, we can characterise concepts by means of classifiers: class, actor (in a use case), and interface are examples of a classifier.

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 and none of its operations have implementations.

An interface is therefore like a very abstract class, a set of common functionalities, typically.

In Java, an interface
- contains only method signature (but not definitions) and
- constants (static and final),
- cannot be instantiated.

An (abstract) class possibly implements one or more interfaces providing definitions for all (or some) methods (mind possible conflicts!).

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 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

- 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

In Java, we say that Rectangle and Balloon implements the interface Moveable. Let us consider some code:

```java
interface Moveable {
    void left(int d);
    void up(int d);
}

class Rectangle implements Moveable {
    ...
}

class Balloon implements Moveable {
    ...
}
```

Class Mover will have a reference to Moveable as an attribute:

```java
class Mover {
    Moveable m;
    ...
    m = new Rectangle(...);
    m.left(17);
    m = new Balloon(...);
    m.up(25);
}
```

The variable m can refer to an object of any class that implements the Moveable interface. So m synthetically represents the feature of being capable of movement, i.e. m.left and m.right.

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. (can this be solved?)
**Interfaces**

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.

Pre- and post-conditions can be attached to an operation in an interface to specify what the implementation of the operation must achieve.

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.

**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`.

**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 `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.

**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 (and by Java).
Together is the UML CASE TOOL that we are going to use in this module.

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.

Packages

A package has no meaning on its own; its behaviour is provided by the classes it contains. However, it does define a namespace.

If different people are working on a problem, then they might use the same name for two different components. If the two components are in different packages, this is OK.

Classes in one package cannot see the classes and services in another package unless the package is imported.

We can model this either as a dependency with stereotype <<imports>> or as a navigable association.