Féidearthachtaí as Cuimse
Infinite Possibilities

- In other words, interfaces primarily define methods that other classes must implement.

Java Interfaces

Object Oriented programming

OLLSCOIL TEICNEOLAÍOCHTA
BHAILE ÁTHA CLIATH
DUBLIN
TECHNOLOGICAL
UNIVERSITY DUBLIN

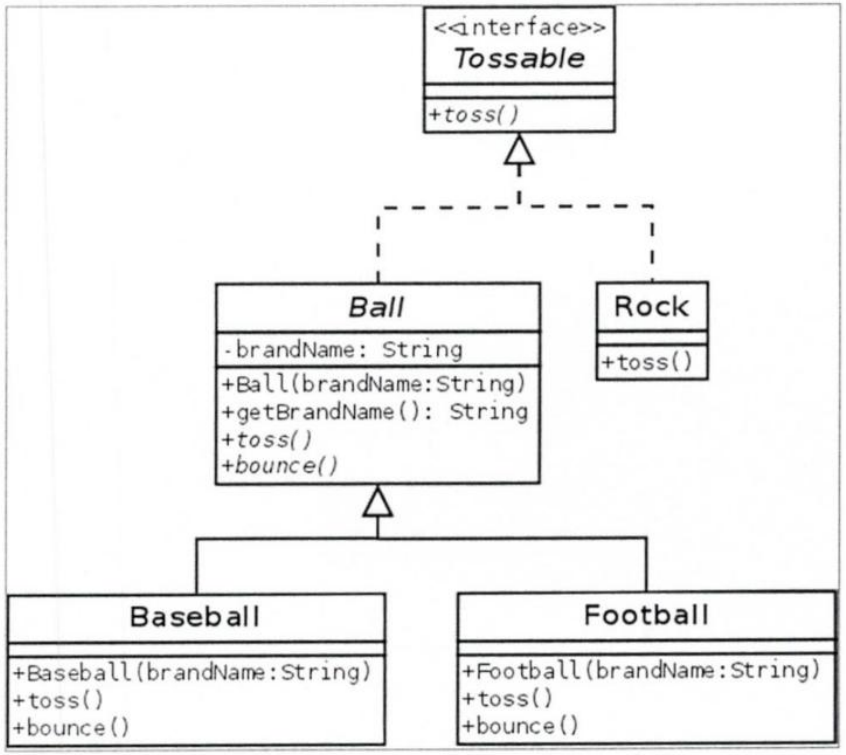
Java - Interfaces

(note: Java interfaces.. Not be mixed up with Graphical user interfaces!)

Learning Outcomes

- What a Java interface is
- Why it is needed
- How to implement it
- UML notation

Question Time
(From Exam Paper 2018)



Section B Figure 1 UML class hierarchy

- What are the superclasses?
- What are the subclasses?
- Can you see method overriding?
- Is this correct?

Class Baseball extends Ball, Rock

- Java **does not support multiple inheritance of classes**.
- BUT
- Java ***does*** support multiple inheritance of interfaces

Scenario

- Supposing you need a set of classes to have specific functionality
  - **BUT WE DON'T WANT TO FORCE THEM INTO THE SAME INHEITANCE TREE– don't share the same parents**

- **Examples Book, Article, Website, DatabaseRecord, all can search(String keyword) but come from very different sources.**

- Need a way to say "these classes implement (and must implement) this/these methods"

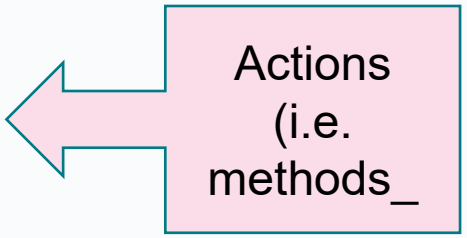
- Achieved through "java interfaces" **Interface Searchable { boolean search(String query); }**

Interfaces (in java)

- An interface is a **description** of a set of behaviour or actions
- A class that implements the interface **must** implement the actions (i.e. methods) declared in the interface.

```
Interface MammalBehaviour
{
    void breathe();
    void speak();

}
```



Syntax

Access Modifiers for Methods in Interfaces

- In Java, all methods declared inside an interface are **implicitly public and abstract**, even if these keywords are not written.
- This means every implementing class must make those methods **public** so they can be accessed and overridden.
- You **cannot** make interface methods private, protected, or package-private — doing so would cause a compiler error, because interface methods are meant to define a public contract.

Interfaces (in java)

```
public class CAT implements MammalBehaviour
{
    ...

  public void breathe()
  {
      System.out.println("breathing...")
  }

  public void speak()
  {
      System.out.println("speaking...")
  }
    ....
}
```

class signs up to implement the i/f

Class (must) implement the behaviour / methods in the i/f

Interfaces and attributes

in **Java**, **interfaces cannot have instance attributes (fields)** in the usual sense (like private String name;) because interfaces don’t hold state.

However, interfaces **can define constants** — that is, **public static final** variables.

An **interface constant** — it’s implicitly public static final:

```
public interface Shape {
    double PI = 3.14159;  // implicitly public, static, and final
}
```

Why are interfaces useful?

- A way to guarantee behaviour across a set of unrelated classes
- A “contract” that the class signs up to..
- A class can implement more than one interface
- If you know a class implements an interface, then you know that class contains concrete implementations of the *methods* declared in that interface, and you are guaranteed to be able to invoke these methods safely.
  (i.e. You know the class can do…)
- **Safe polymorphism**

How to know if a class implements a particular Interface?

- Previous example:

// if we have a cat object:

```
Cat c2 = new Cat ("fluffy", 10);
```

c2 object is of type "Cat";

c2 object is also of type MamalBehaviour

If (c2 `instanceof` MamalBehaviour)...

    // know c2 can breathe() and speak()

    // i.e. `c2.breathe(), c2.speak()` safe to call

Similar to Abstract Class--

- You can create a *reference variable* of an **interface type**, and assign to it an **object of any class** that implements that interface.

```
MammalBehaviour pet = new Dog();
 pet.breathe();
 pet.speak();
```

- **polymorphism via interfaces**.
  - If a class implements an interface,
  - you can use the interface name as the **type** of the variable that refers to that object.
  - **But —**

    you can't actually make a *new interface object* (because interfaces can't be instantiated) --- like abstract class can't be instantiated

Java Polymorphism & instanceof Example

A variable's **declared type** decides what methods you can *call*,
but the object's **actual type** decides *which version* of those methods runs at **runtime**.

```
 MammalBehaviour pet = new Dog();

pet.breathe();   // OK (in interface)

pet.speak();     // OK (in interface)

// pet.fetch();  // ❌ Not allowed - not in MammalBehaviour
```

```
MammalBehaviour

System.out.println("Type: " + pet.getClass().getSimpleName());

if (pet instanceof Dog) {

    System.out.println("pet is a Dog");

    Dog d = (Dog) pet;      // downcast

    d.wagTail();            // ✅ now allowed

}
```

Note: Can contain non empty methods

- As of Java 8, "`default`" methods introduced
- All classes implementing the i/f inherit this behaviour (but can override)

```
interface IMammal
{
    void breathe();
    void spreak();

    default void sleep()
    {
       System.out.println("sleeping");
    }
}
```

Why Default Methods

Default methods were introduced so that interfaces could **evolve** without breaking older code.

For example:

- Suppose hundreds of classes implement your interface MammalBehaviour.
- Later, you want to add a new method (e.g. sleep()).
- Without default methods, **all those classes (Dog,Cat) would stop compiling**, because they'd all need to implement the new method.
- With a default method, you can give it a *standard implementation*, and existing classes automatically inherit it.

Example:

- Take the `Car, Vehicle etc` classes from Lab 4

  - Implements an interface – called… (just a made up name…)

    `VehicleFunctions`

VehicleFunctions interface

```
public interface VehicleFunctions {
    void startEngine();
    void stopEngine();
    int getTopSpeed();
}
```

Example:

- Note:
- “`instanceof` “ test the “type” of a class.

Some rules

- You can’t instantiate an Interface
- All methods are implicitly public
- Where a class implements an interface instances of the class can be treated as objects of the interface type as well as objects of their own class type

Why are interfaces useful? (again)

- A way to guaranteeing behaviour across a set of unrelated classes
- If you know a class implements an interface, then you know that class contains concrete implementations of the methods declared in that interface, and you are guaranteed to be able to invoke these methods safely.
- Some interfaces are useful for “tagging” classes.. An interface with no methods in it is referred to as a **tagging** interface
  - `.e.g Serializable interface`

An example of a tagging interface: Serializable

- Allows (values of) an object to be transformed into bytes (and back again)

- to transfer and store objects persistently
  - e.g. ATM withdrawl

- Simply interface the “Serializiable” interface .. to make class of type serializable. No methods

- `Class person implements serializable`

Further Explanation

If you need to:

- save an object to a file,
- send it across a network,
- store it in a database, or
- pass it between activities in Android,

then you need that class to be *serializable*.

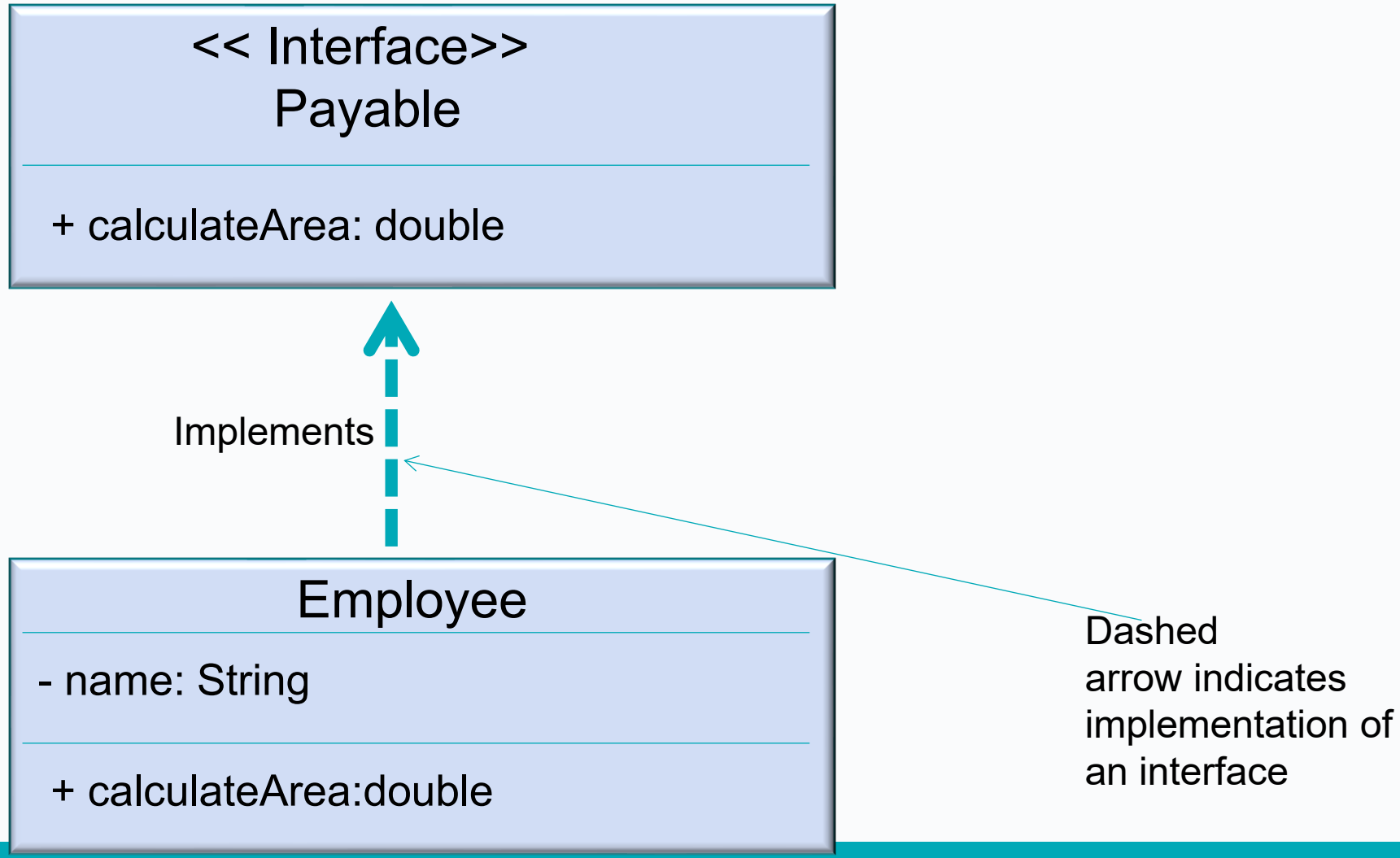
Interfaces can inherit from each other

- `public` **`Interface payable,`** `implements writeable, printable`

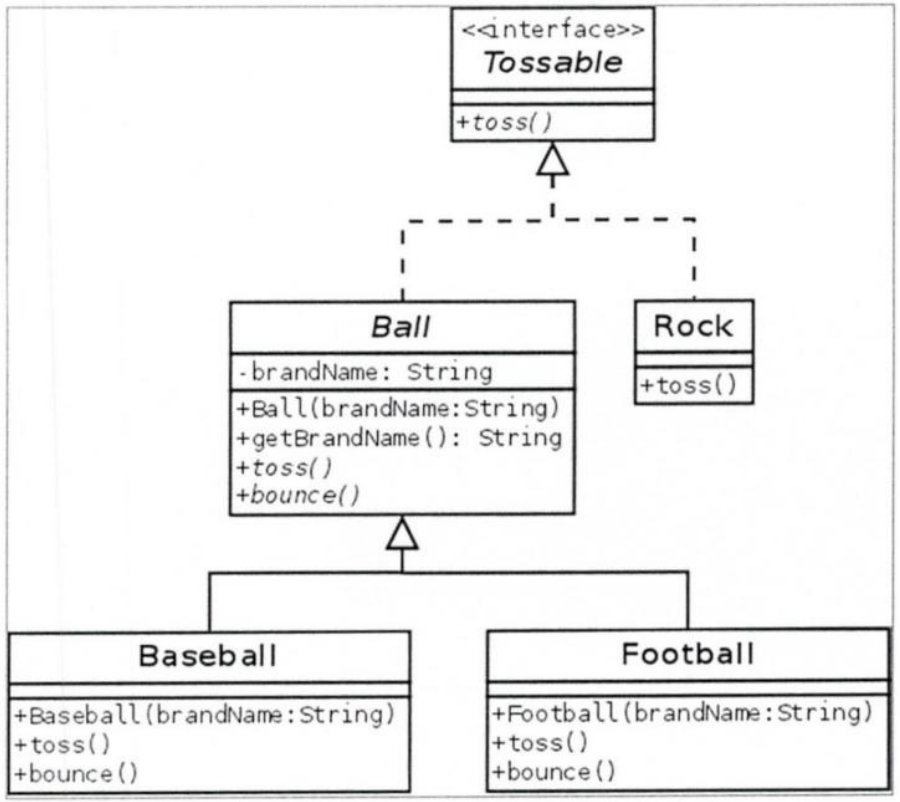
- All methods in writeable, printable interfaces are in the Payable interface...

- A class that implements Payable has to implement all the inherited methods in payable too...

UML: interfaces notation



Writing an interface



Section B Figure 1 UML class hierarchy

Rules

- Start with the- **access modifier** then keyword **interface** (lowercase) then name of interface in Uppercase, followed by method – return type and name.

```
public interface Tossable
{
//method
 void toss()
}
```

Summary

The purpose of interfaces is to ensure that particular behaviour is provided in a class that implements that interface. All empty methods listed in the interface must be implemented in the class

What we covererd

- What a Java interface is
- Why it is needed
- How to implement it
- UML notation