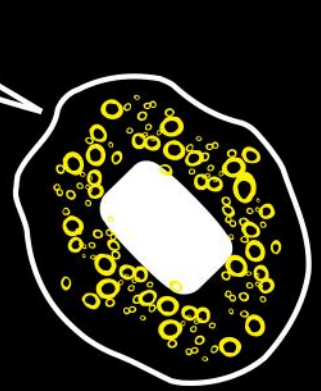
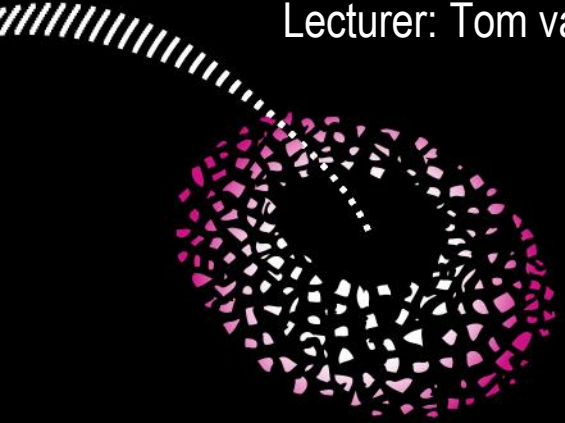
UNIVERSITY OF TWENTE.

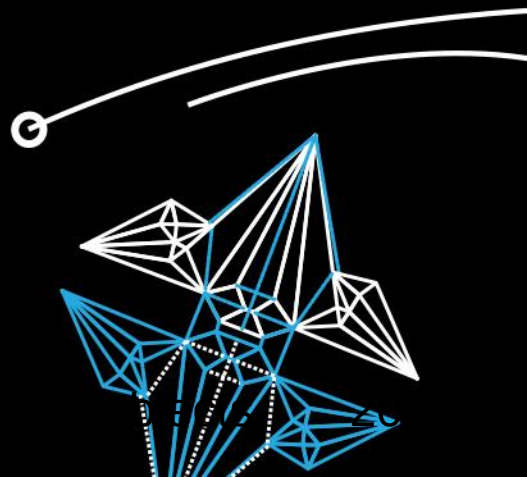


Interfaces and Abstract Classes

Topic of Software Systems (TCS module 2)

Lecturer: Tom van Dijk





ABSTRACT METHODS AND CLASSES

An abstract method is a method without a body

```
public abstract class SomeAbstractClass {
    public abstract void doSomething(int someNumber);
    protected abstract double computeSomething(int numberOne, double numberTwo);
}
```

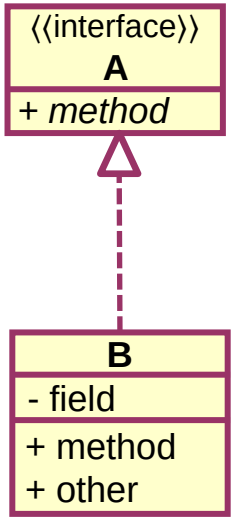
- Abstract methods must be in an abstract class
- Abstract classes are incomplete, partial, unfinished
- Subclasses must either implement the abstract methods or also be abstract
- You cannot instantiate an abstract class but an abstract class has a constructor (for subclasses)

INTERFACES

An interface is a special type of class: only specification, no implementation

- All fields are implicitly public final static (constants)
- All methods are implicitly public abstract

- Classes can extend one class, and implement multiple interfaces
- Interfaces can extend multiple interfaces



INTERFACES

Very simple syntax

```
interface MyInterface {
    /**
     * Specification 1 ...
     */
    void myMethod1();   // <-- a semicolon instead of the method body

    /**
     * Specification 2 ...
     */
    int myMethod2(int i, int j);
}
```

INTERFACE IMPLEMENTATION: EXAMPLE

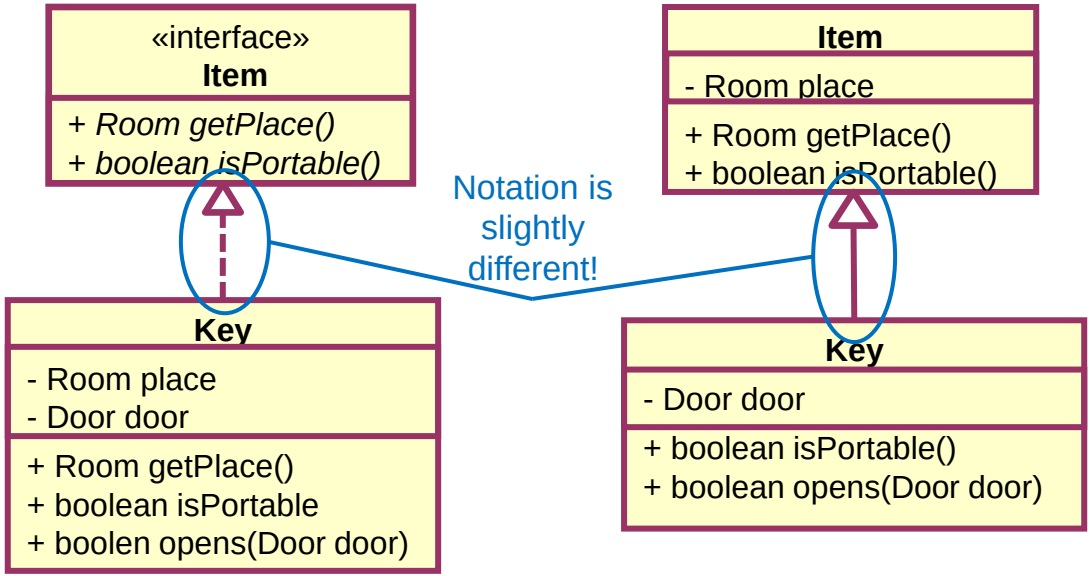
```
interface Item {
    Room getPlace();
    boolean isPortable();
}
```

```
public class Key implements Item {
   private Room place;
   private Door door;

   public Key(Door door) {
      this.door = door;
   }

   public Room getPlace() {
      return place;
   }
   public boolean isPortable() {
      return true;
   }

   public boolean opens(Door door) {
      return this.door.equals(door);
   }
}
```

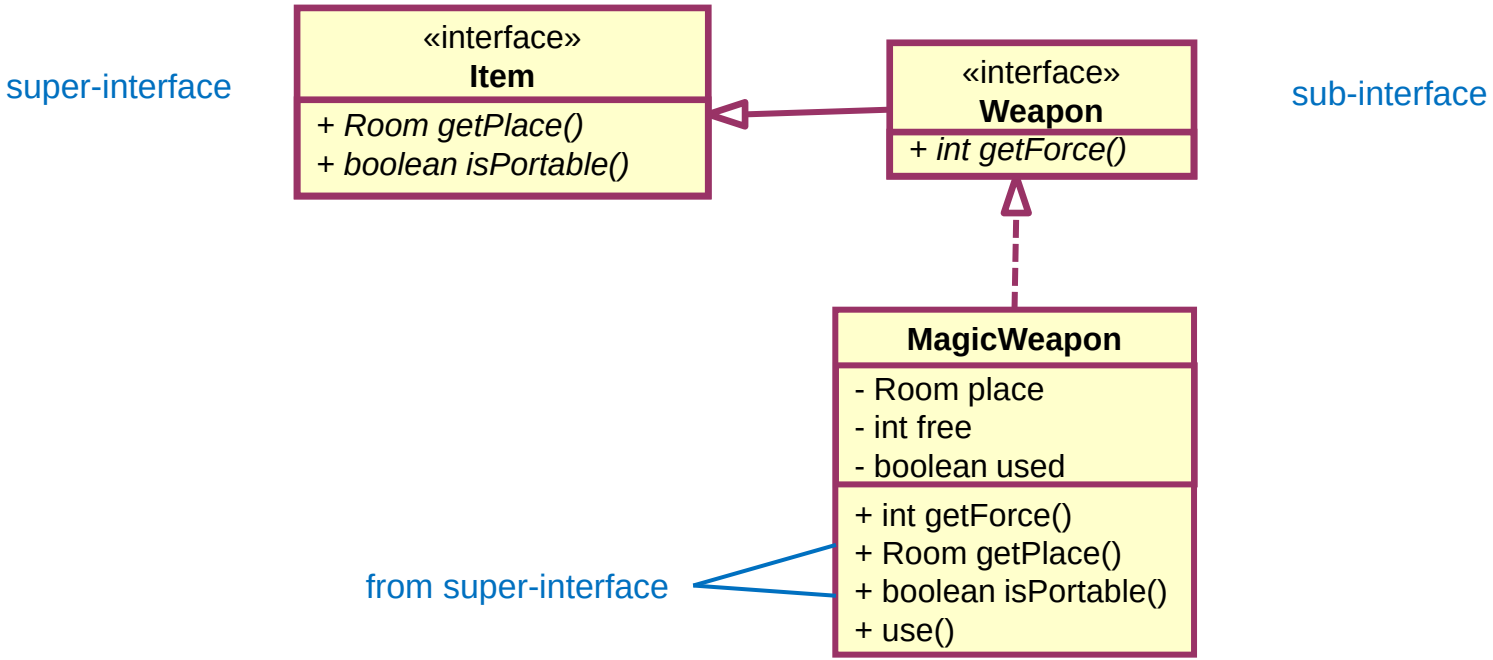
implemented methods have same signature (method names, result, parameter types) and visibility

implementation has additional methods, fields and constructor

NOTATION


«interface»
Item
+ Room getPlace()
+ boolean isPortable()
Key
- Room place
- Door door
+ Room getPlace()
+ boolean isPortable
+ boolen opens(Door door)
Notation is slightly different!
Item
- Room place
+ Room getPlace()
+ boolean isPortable()
Key
- Door door
+ boolean isPortable()
+ boolean opens(Door door)

COMBINING INTERFACES AND INHERITANCE



INTERFACES

Example of multiple inheritance

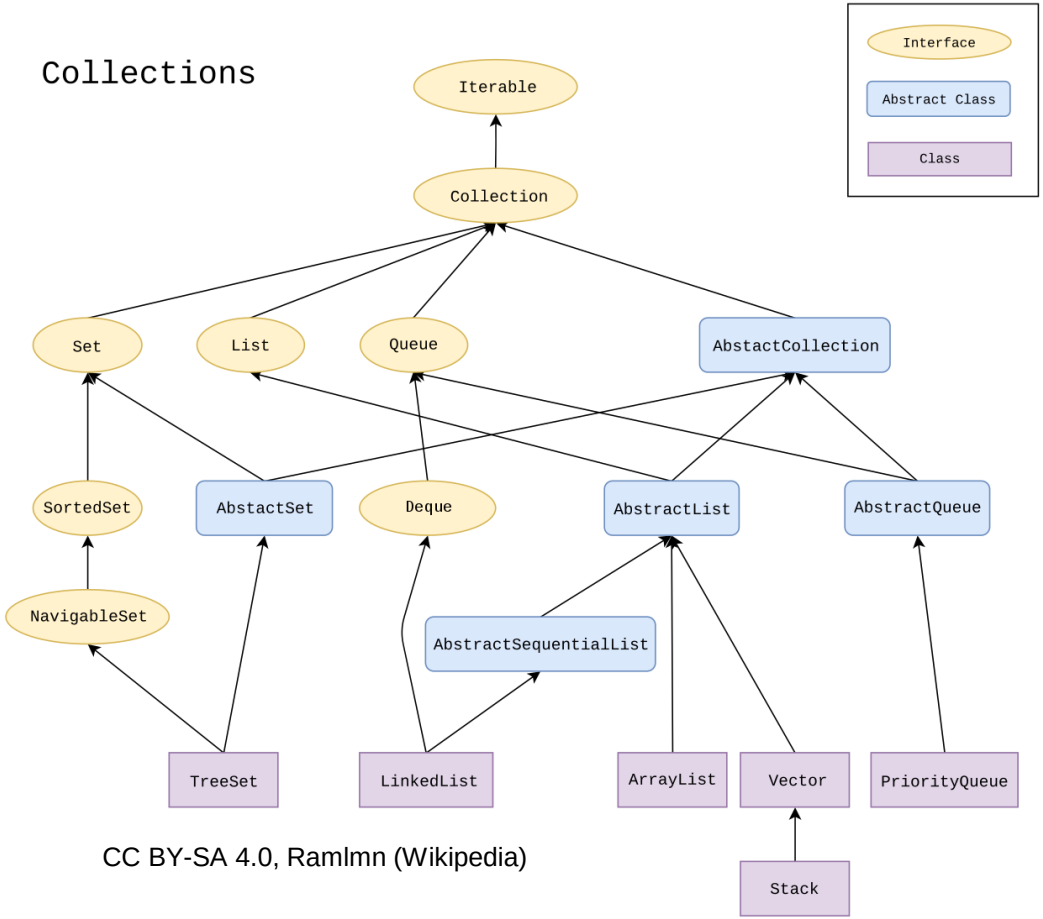
```
interface Stove {
    void on(int temperature);
    void off();
}

interface Microwave {
    void nuke(int watt, int duration);
}

public class CombiOven implements Stove, Microwave { ... }
```

INTERFACES

- The Collections library includes many useful datastructures including list types
- All list types have a common interface List
- List defines many methods. Some of the more important ones are:

```
interface List {
    boolean add(Object e);
    boolean remove(Object e);
    boolean contains(Object e);
    Object get(int index);
    Object set(int index, Object e);
    Object remove(int index);
    int size();
}
```

Collections

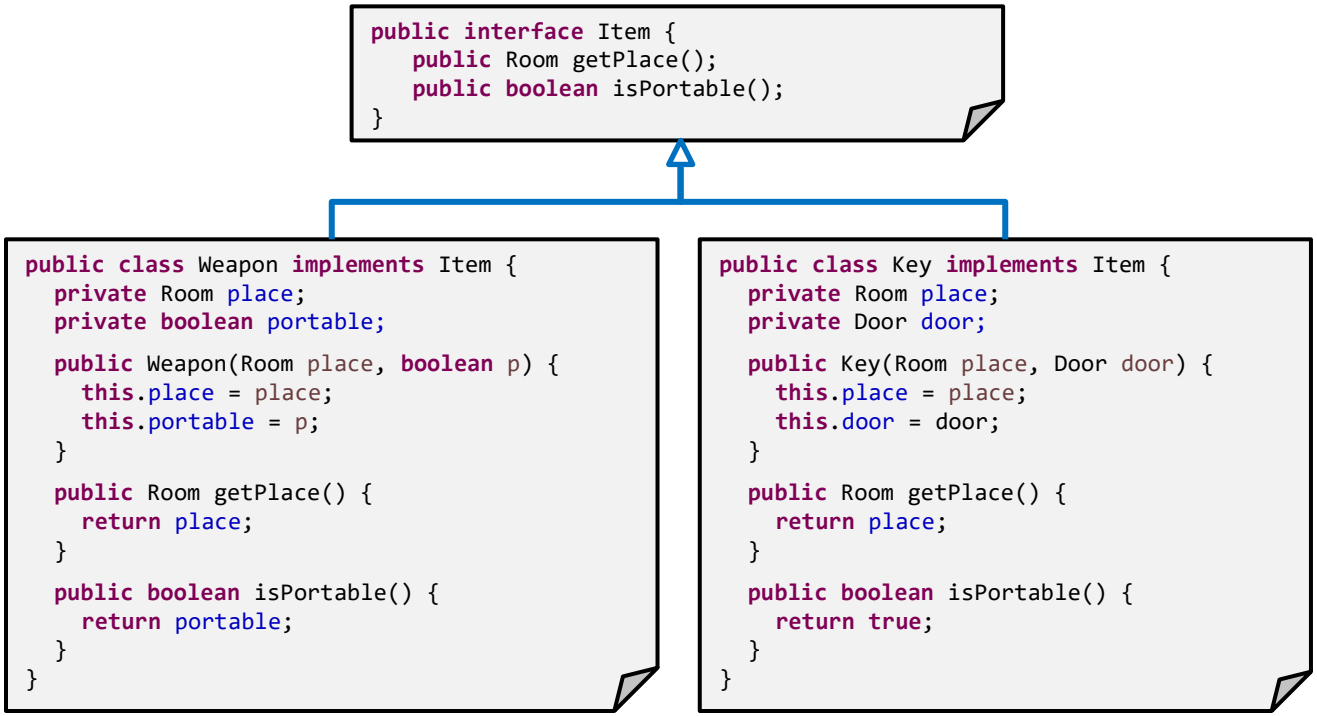


EXAMPLE INTERFACE: JAVA.UTIL.COMPARABLE

```
interface Comparable {
    /**
     * @return negative, zero, or positive if this
     *   object is less than, equal to, or greater than o
     */
    int compareTo(Object o);
}
```

- Implemented by java.lang.String: alphabetical order
  - `class String implements Comparable`
  - What is the result of "this".compareTo("that")?
- Implemented by java.util.Date: temporal order
  - `new Date(2013,11,26).compareTo(new Date(2014,11,26))`?

ABSTRACT CLASSES: EXAMPLE

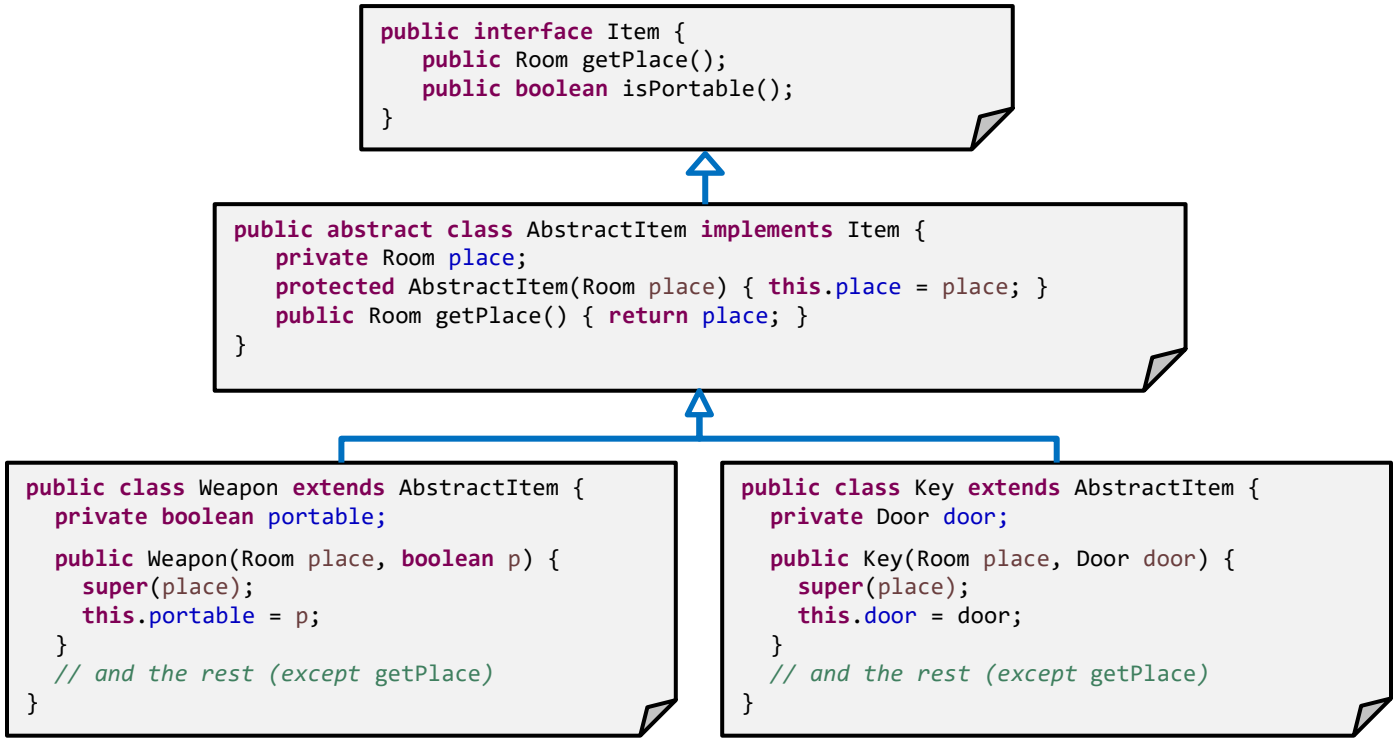


```java
public interface Item {
    public Room getPlace();
    public boolean isPortable();
}
```

```java
public class Weapon implements Item {
  private Room place;
  private boolean portable;

  public Weapon(Room place, boolean p) {
    this.place = place;
    this.portable = p;
  }

  public Room getPlace() {
    return place;
  }

  public boolean isPortable() {
    return portable;
  }
}
```

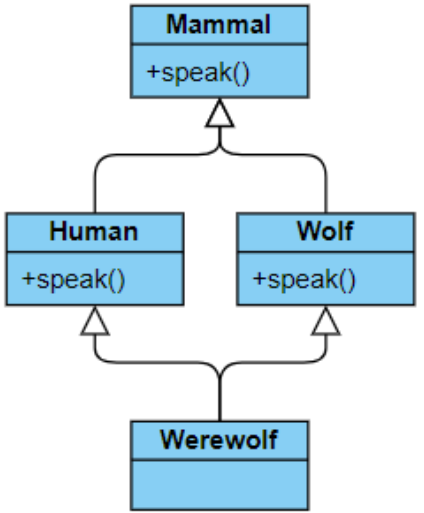
```java
public class Key implements Item {
  private Room place;
  private Door door;

  public Key(Room place, Door door) {
    this.place = place;
    this.door = door;
  }

  public Room getPlace() {
    return place;
  }

  public boolean isPortable() {
    return true;
  }
}
```

ABSTRACT CLASSES: EXAMPLE (CONTINUED)



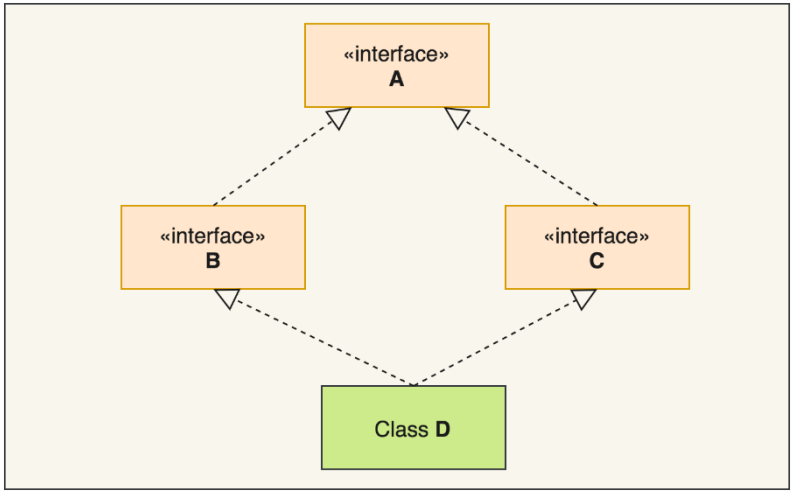
MULTIPLE INHERITANCE

- In many programming languages, a class can extend multiple classes
- This leads to the famous diamond problem



MULTIPLE INHERITANCE IN JAVA

- In Java, classes can implement multiple interfaces



- However, interfaces have no method body so no problem!

DEFAULT METHODS IN INTERFACES

Java 8 introduces static methods and default methods to interfaces

```
interface A {
   public int getANumber();
}
```

```
public class B implements A {
   private int theBestNumberFolks;

   public int getANumber() {
      return theBestNumberFolks;
   }
}
```

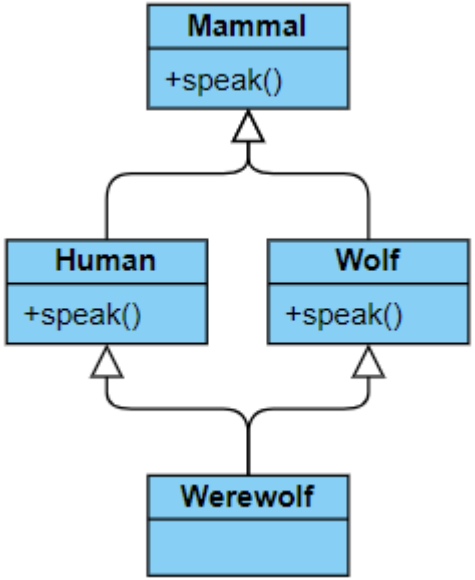
```
interface A {
   default public int getANumber() {
      return 42;
   }
}
```

…. oops? The diamond problem is back!

MULTIPLE INHERITANCE

What if speak() is a default method in Human and Wolf?

Which implementation does Werewolf get?



DEFAULT METHODS IN INTERFACES

Conflict resolution (= which same-signature method does the class inherit??)

1. Method inherited from the superclass take priority over default interface methods
2. Methods from subinterfaces take priority over the superinterfaces
3. Error! The implementing class must provide its own implementation!

BENEFITS OF INTERFACES

- No default implementation of methods necessary
- Classes can implement multiple interfaces
  - Possibly combined with extending one class
  - Default methods are inherited from each interface

```
public class Wand implements Weapon, Magical extends WoodenItem {
    // class body
}
```

BENEFITS OF INHERITANCE OVER INTERFACES

- More reuse of methods (default methods in interfaces are limited)
- Classes and abstract classes can have non-final fields
- Classes can have protected and private members

INHERITANCE VS INTERFACES

Abstract classes are a “basis for subclasses with shared behaviour”

Interfaces are specifications, describing the behaviour of an implementing class

Often, abstract classes implement interfaces, and other classes use the interfaces

INHERITANCE VS COMPOSITION

Inheritance: inherit fields and methods from another class

- Use when there is a clear parent-child relationship of concepts (is-a relationship)
- Use to alter the behaviour of a class
- Use when you want to reuse the entire interface of the superclass

Composition: rely on other object(s) to provide (some) functionality

- Composition is often more appropriate
- Use when only using parts of the functionality of another class (has-a or uses-a relationship)

Both are fundamental in object-oriented programming!

INHERITANCE VS COMPOSITION

Example:

- Use inheritance if you make a List that is a specialised standard List
- Use compisition if your class is not a List but uses a List