3.5 Interfaces

Software and Analysis Design

2nd Year, Computer Science

Universidad Autónoma de Madrid

Interfaces: What are they?

- A class provides an interface by offering methods to manipulate its instances
- An interface is a group of related method signatures (headers only), modelling the behaviour of a type of objects
- In Java, an interface consists of method declarations (not necessarily implemented) and constants
- There are also empty interfaces: Used as semantic labels that can be associated with certain objects, e.g., `Serializable`, `Cloneable`

Why are they needed?

- An interface defines a type
  - Similar to a class but providing only *public abstract* methods and *public constants* (public static final)
- An important difference regarding multiple inheritance
  - Java does not permit multiple inheritance with classes
  - An interface may inherit from zero o more interfaces, but not from classes
- A class may extend only from one class, and it may additionally *implement* any number of interfaces

Example

```
public interface Job{
  void executeJob();
}

public class MakeBackup
       implements Job
{
  Database d;
  public void executeJob(){
    // Code to implement backup
  }
}
```

```
public class JobQueue
       implements Job {
  List<Job> pendingJobs;

  public void addJob(Job j){
    pendingJobs.add(j);
  }

  public void executeJob(){
    for (Job j:pendingJobs){
      j.executeJob();
    }
    pendingJobs.clear();
  }
}
```

Example

"implements" means that the class provides an implementation for one or more interfaces

```
public interface Job{
  void executeJob();
}

public class MakeBackup
       implements Job
{
  Database d;
  public void executeJob(){
    // Code to implement backup
  }
}
```

```
public class JobQueue
       implements Job {
  List<Job> pendingJobs;

  public void addJob(Job j){
    pendingJobs.add(j);
  }

  public void executeJob(){
    for (Job j:pendingJobs){
      j.executeJob();
    }
    pendingJobs.clear();
  }
}
```

Example

```
public interface Job{
  void executeJob();
}

public class MakeBackup
       implements Job
{
  Database d;
  public void executeJob(){
    // Code to implement backup
  }
}
```

```
public class JobQueue
       implements Job {
  List<Job> pendingJobs;

  public void addJob(Job j){
    pendingJobs.add(j);
  }

  public void executeJob(){
    for (Job j:pendingJobs){
      j.executeJob();
    }
    pendingJobs.clear();
  }
}
```

Both classes must implement the methods declared in the interface

Example

```
public interface Job{
  void executeJob();
}

public class MakeBackup
       implements Job
{
  Database d;
  public void executeJob(){
    // Code to implement backup
  }
}
```

```
public class JobQueue
       implements Job {
  List<Job> pendingJobs;

  public void addJob(Job j){
    pendingJobs.add(j);
  }

  public void executeJob(){
    for (Job j:pendingJobs){
      j.executeJob();
    }
    pendingJobs.clear();
  }
}
```

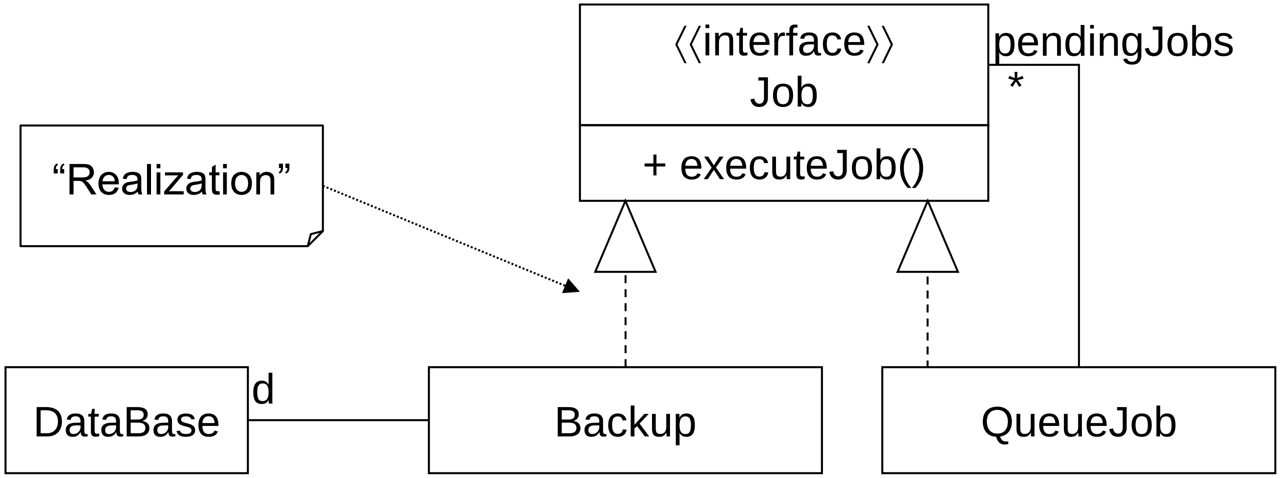
Job is a type
Any object of a class implementing that interface is considered an instance of that type

Declaring interfaces: Syntax

```
[public | ...]? interface <interface-name>
        [extends <interface-name1>,
                    <interface-name2> ... ]? {
        <interface-members>
}
```

- An interface may extend (inherit) from other interfaces
- Variables in an interface:
  - `[public static final] <type> <variable-name>=<value>;`
  - Implicitly they are all public static final variables
- Methods are declared without implementation (akin to abstract methods)
  - Any class implementing the interface must provide implementations for all interface methods (or else be declared abstract)
  - Keywords `public` and `abstract` can be specified but are optional since they are assumed

Interfaces in UML



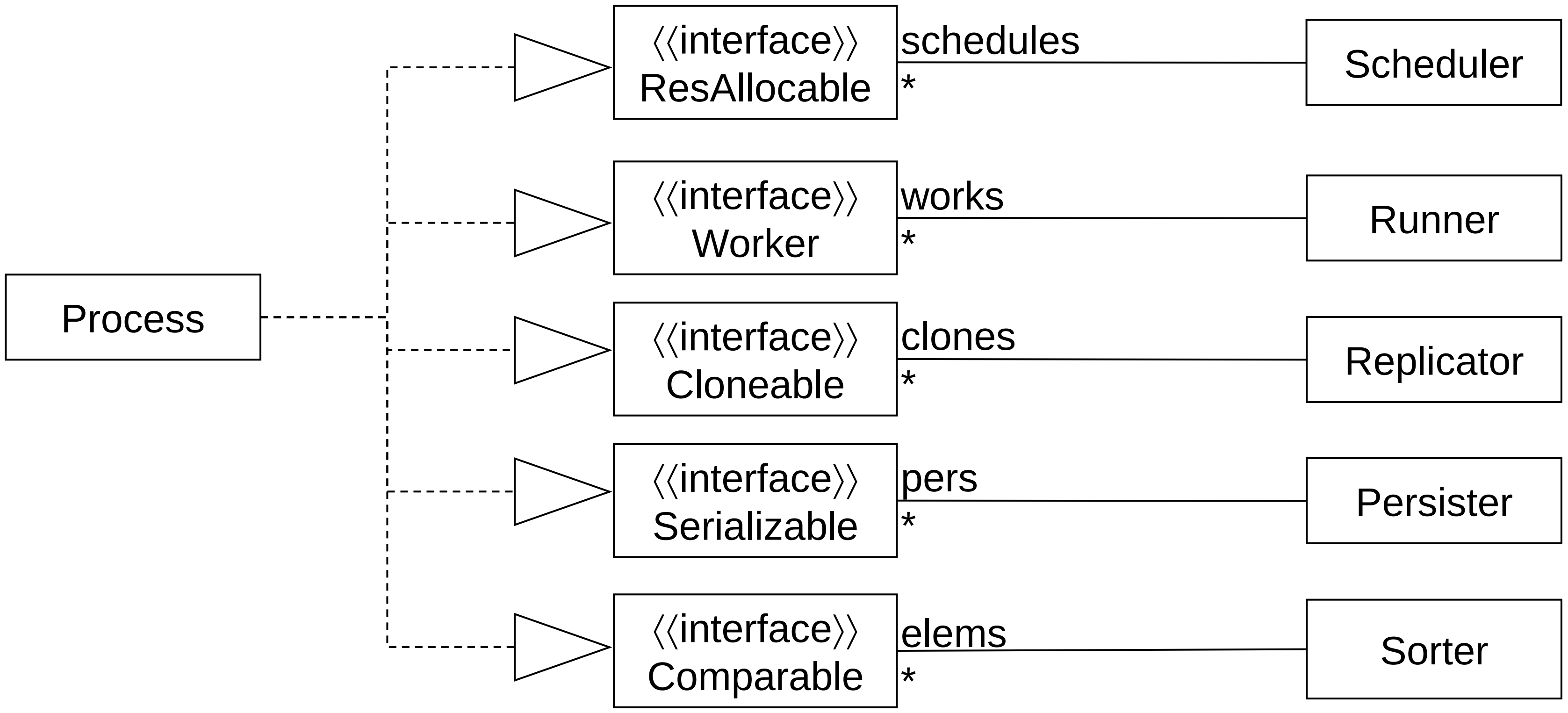
Exercise

- Design the API of an utility class with a sorting method called `sort`, to sort lists of strings

- The `sort` method should sort the list elements by an ordering criteria to be defined by the user, and which is passed as parameter
  - As an example, define a criteria based on the String length

Usefulness

- Interfaces allow defining functionality that facilitates accessing (possibly unrelated) classes homogeneously
- They permit looking at a class from different perspectives



Example

Comparable interface

```
public interface Comparable{
  int compareTo(Object o)
}

public class Car
implements Comparable
{
  int power;
  String marca;
  String modelo;
  //…
  int compareTo (Object o){
    Car c= (Car) o;
    return power - c.power;
  }
}
List aList;
// …
Collections.sort(aList);
```

- Definided in `java.lang`.
- Defines the ***natural ordering*** of a class.
- Objects from classes implementing it can be sorted by using, for example, `Collections.sort` and `Arrays.sort`

Problem: Two classes implementing this interface may not be directly comparable with each other.

Example: Generic Interfaces

Interface Comparable<T>

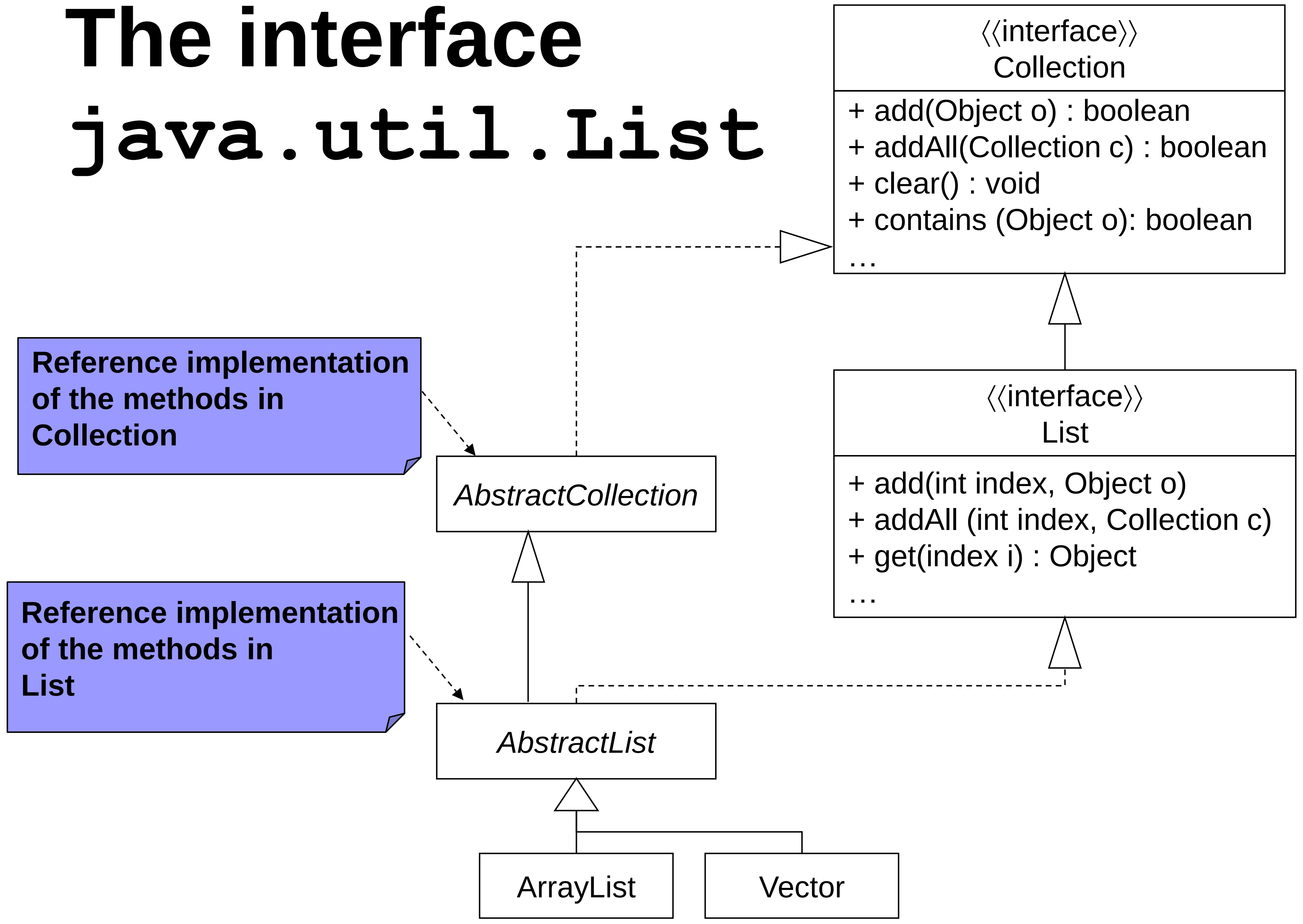
```
public interface Comparable<T>{
  int compareTo(T o)
}

public class Car
implements Comparable<Car>
{
  int power;
  String brand;
  String model;
  // …
  int compareTo (Car c){
    return power - c.power;
  }
}
List<Car> sportsCars;
…
Collections.sort(sportsCars.sort);
```

- Generic Interface new since Java 5.0
- Avoids problems regarding comparison of incompatible data types

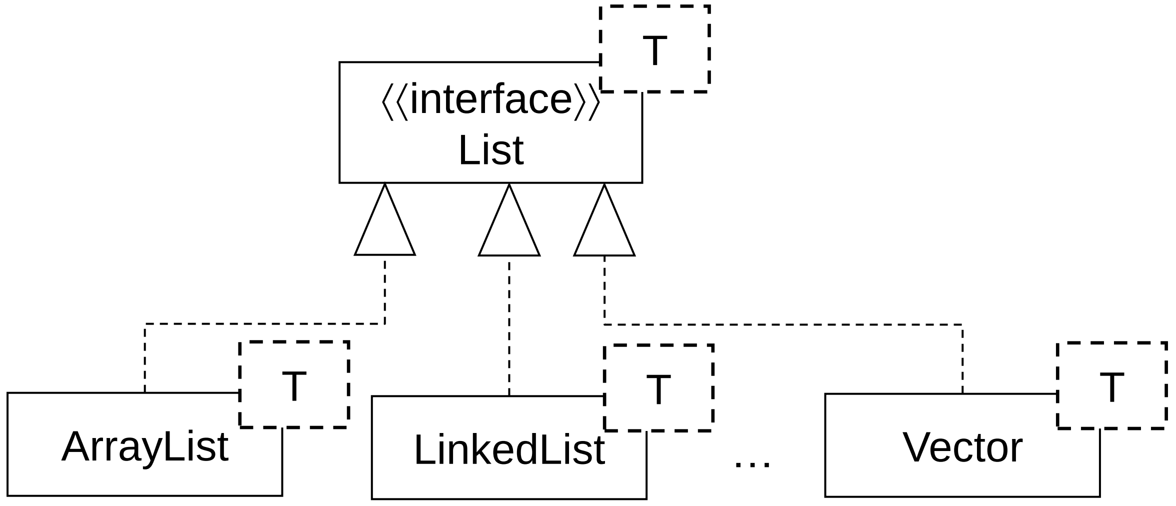
Interfaces in standard libraries

- *Cloneable*: States that the method `Object.clone()` may be used to copy the object by copying all its components
- *Comparable<T>*: An object can be compared with another object of type T
- *Comparator<T>*: Provides a function to compare two objects of type T
  - int compare(T o1, T o2)
- *Serializable*: Instances can be serialized (copied in binary format onto a disk or sent through a network)
- ADTs: Collection, Iterable, Queue, Deque, List, Map, Set, TreeModel
- DOM: Document, Node, Element, Attr

The interface `java.util.List`


⟨⟨interface⟩⟩
Collection
+ add(Object o) : boolean
+ addAll(Collection c) : boolean
+ clear() : void
+ contains (Object o): boolean
…
⟨⟨interface⟩⟩
List
+ add(int index, Object o)
+ addAll (int index, Collection c)
+ get(index i) : Object
…
Reference implementation of the methods in Collection
AbstractCollection
Reference implementation of the methods in List
AbstractList
ArrayList
Vector

Example: List<>



- All objects of type List<> can be used in a uniform way (we do not care about the concrete class we are using)

```
void method(List<String> l) {…}
```

- We can change the implementation class without changing the client code

```
obj.method(new ArrayList<String>() )
obj.method(new LinkedList<String>() )
…
```

Modification of interfaces

- Once an interface is created and some classes implement it, if we add methods to the interface or modify them, it is necessary to change all classes that implemented it.

- Solutions?
  - Create a new interface that extends the original one and also includes the new methods
  - Provide a *reference implementation* in form of an abstract class (problem: Java does not allow inheritance from multiple classes)
  - Provide a *reference implementation* in form of an interface with *default* methods (new in Java 8)

Example of modification

```
public interface Job{
  void executeJob();
}

public interface TransactionalJob extends Job{
  void transactionalExecution(Transaction t);
}
```

- The new interface offers an extended functionality, and this does not require modifying the classes implementing the reduced version

- **Problem**: Previous classes do not offer the new interface, although the new methods can be implemented using the previous ones

Reference Implementations

- They help simplifying the definition of interfaces that have many methods, some of which can be implemented by calling other previously implemented methods.

- An example from JDK: `java.util.AbstractList`
  - To have a read-only list, we only need to implement two methods: *get(int)* and *size()*
  - It provides many methods: *iterator, equals, indexOf(Object), lastIndexOf(Object), subList()*, etc.

Create a *Reference Implementation*

```
public interface Tree {
  Object getElement();
  Tree leftChild();
  Tree rightChild();
  boolean isLeaf();
  boolean isEmpty();
  Object search(Object o);
}
```

Subclasses of AbstractTree overwrite some methods to provide implementations, or sometimes for efficiency reasons.

```
public abstract class AbstractTree
implements Tree
{
  public boolean isLeaf(){
    return leftChild().
           isEmpty()&&
           rightChild().
           isEmpty(); }
  public Object search(Object o){
  //implement binary search
  }
  // We can implement other methods
  // by simply raising an exception
}
```

Multiple inheritance with interfaces

```
class C ~~extends~~ A, B{
}
```

- Not allowed in Java!
- Instead we can use interfaces:

```
interface A{}

interface B{}

interface C extends A, B{}

class AImpl implements A{ }

class BImpl implements B{ }
```

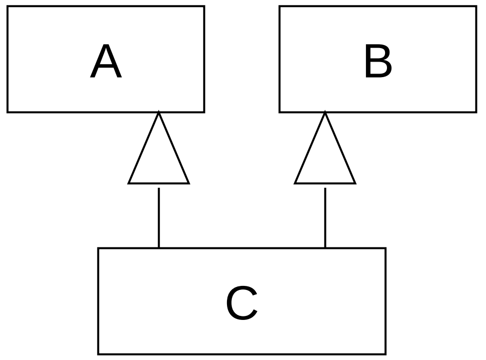
- Use inheritance from the more complex class and delegation with the other class:

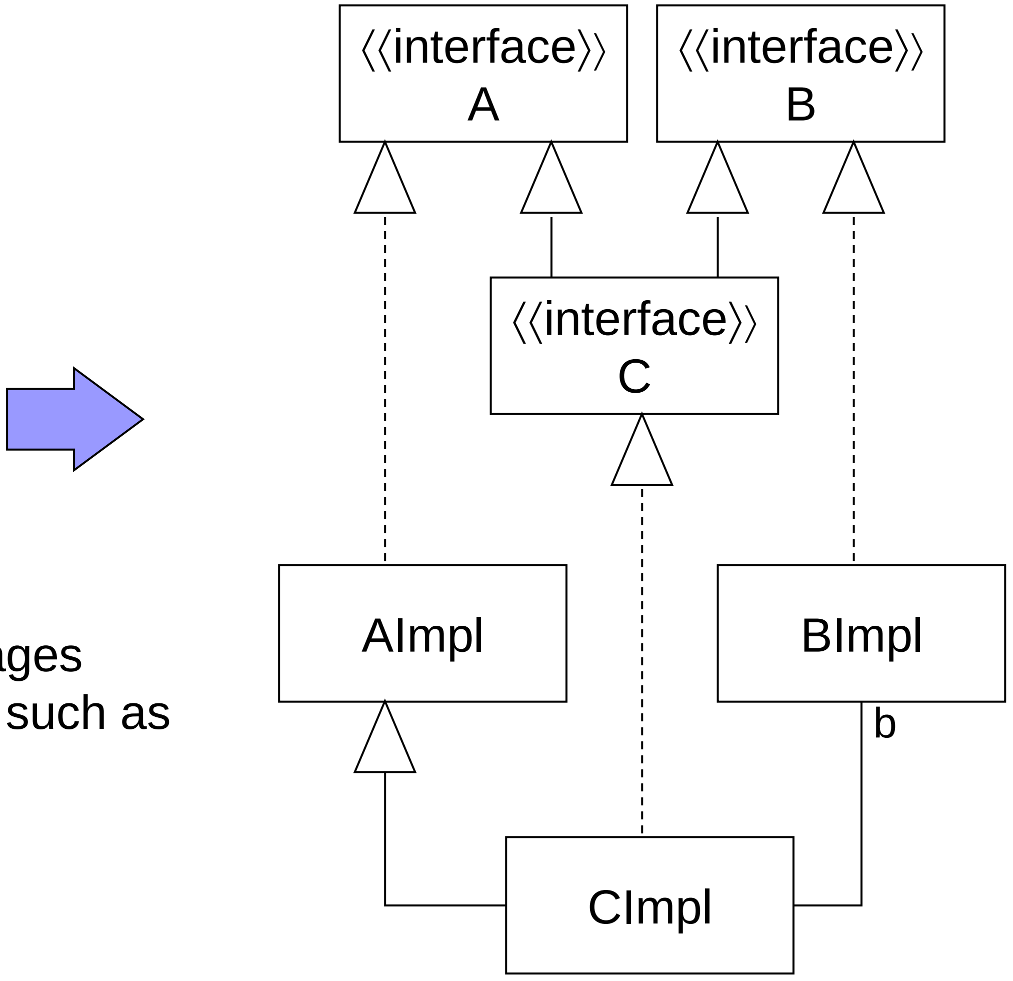
```
class CImpl extends AImpl
            implements C
{
  B b= new BImpl();
}
```

- Delegate in «b» all methods from class B while methods from A are inherited. For example:

```
  public int bIntMethod() {
      return b.bIntMethod();
   }
```

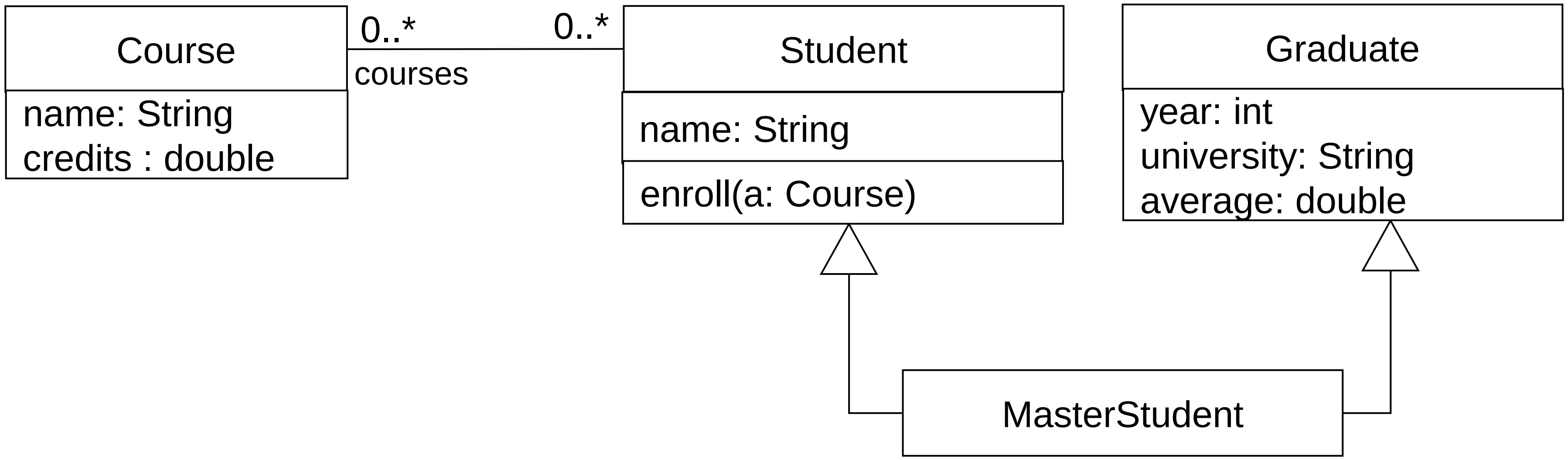
Multiple inheritance with interfaces



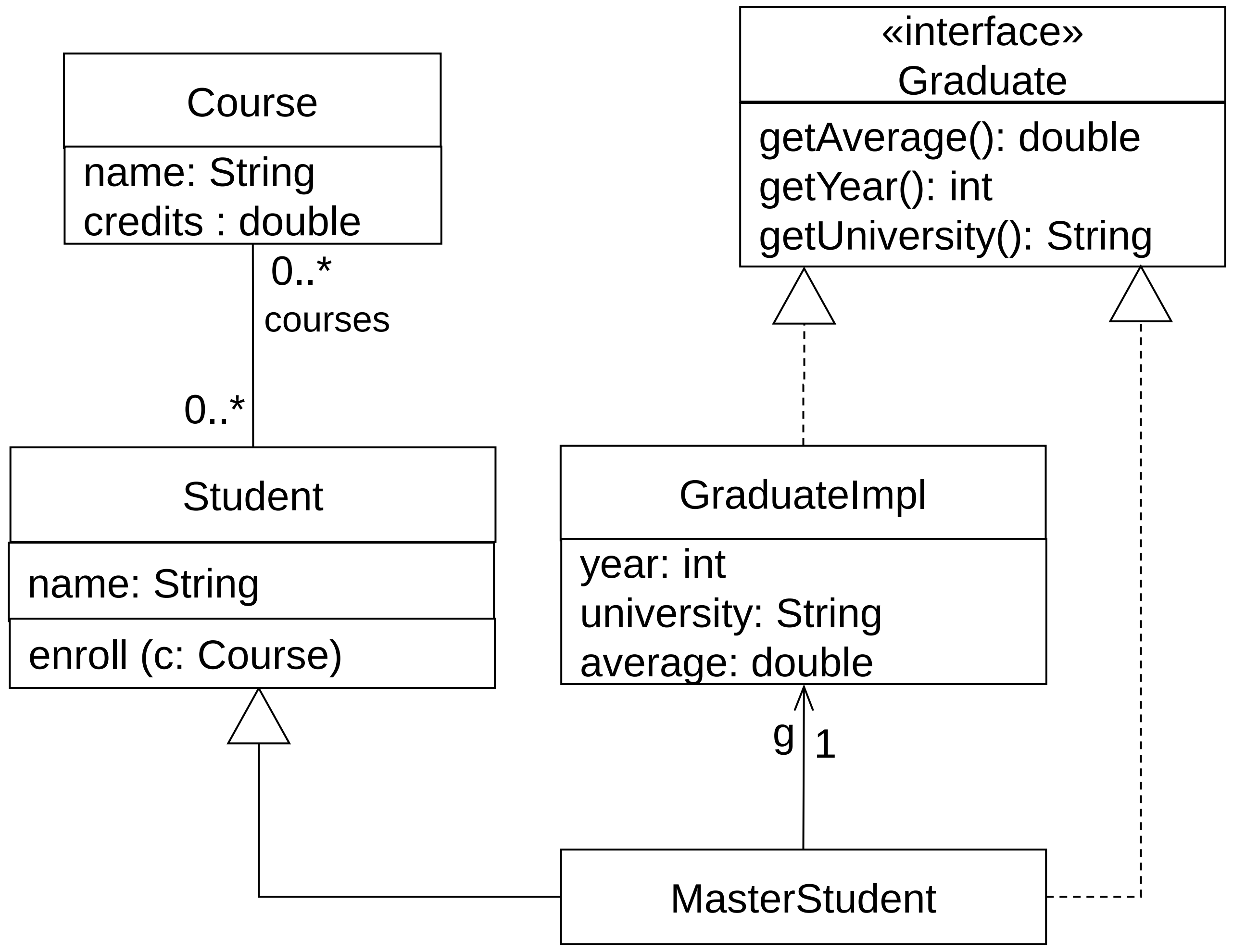


This is valid in UML and languages supporting multiple inheritance such as C++, but not in Java

Exercise (i)


Course
name: String
credits : double
0..*
courses
0..*
Student
name: String
enroll(a: Course)
Graduate
year: int
university: String
average: double
MasterStudent

Exercise (i, solution)



Exercise (ii)

- Build an utility class `PrettyPrinter` to print tree-like structures using indentation

- The utility class should by highly reusable, e.g., with the `Folder` class of the previous exercise

Summary

- An interface defines the protocol to be used for communication among objects.
- An interface consists of method declarations (not necessarily implemented) and constant declarations.
- A class that implements an interface must provide an implementation for all methods in the interface (or else must be declared abstract)
- An interface defines a type: its name can be used anywhere a type is expected