SWINBURNE UNIVERSITY OF TECHNOLOGY

OOP

Abstract classes and Interfaces

Learning Objectives

Students should be able to:

- describe what an abstract class is
- describe what an interface is
- explain why an interface might be used
- implement an interface and use an interface
- describe when interfaces should be used
- understands the role of an interface as a type

Abstract classes

- Sometimes there will be classes which you never intend to create objects…
- Abstract classes cannot be instantiated but can be subclassed
- They are used as super classes in an inheritance hierarchy
- The purpose is to provide an appropriate super class from which other classes can inherit, thus share a common design
- A class declared as abstract may or may not include abstract methods. And if a class includes abstract methods it MUST be declared as abstract.

□ Abstract Method: A method with no implementation (no method body)

□ Abstract methods are meant to be overridden by sub classes.

Abstract classes

- The abstract class represents the generic or abstract form of all the classes that are derived from it.
- A class becomes abstract when you place the abstract key word in the class definition.

```
public abstract class ClassName
```

SWINBURNE UNIVERSITY OF TECHNOLOGY

Abstract Methods

- An abstract method has no body and must be overridden in a subclass.
- An *abstract method* is a method that appears in a superclass, but expects to be overridden in a subclass.
- An abstract method has only a header and no body.

```
AccessSpecifier abstract ReturnType
   MethodName(ParameterList);
```

- Example:
  - □ Student.java, CompSciStudent.java, CompSciStudentDemo.java

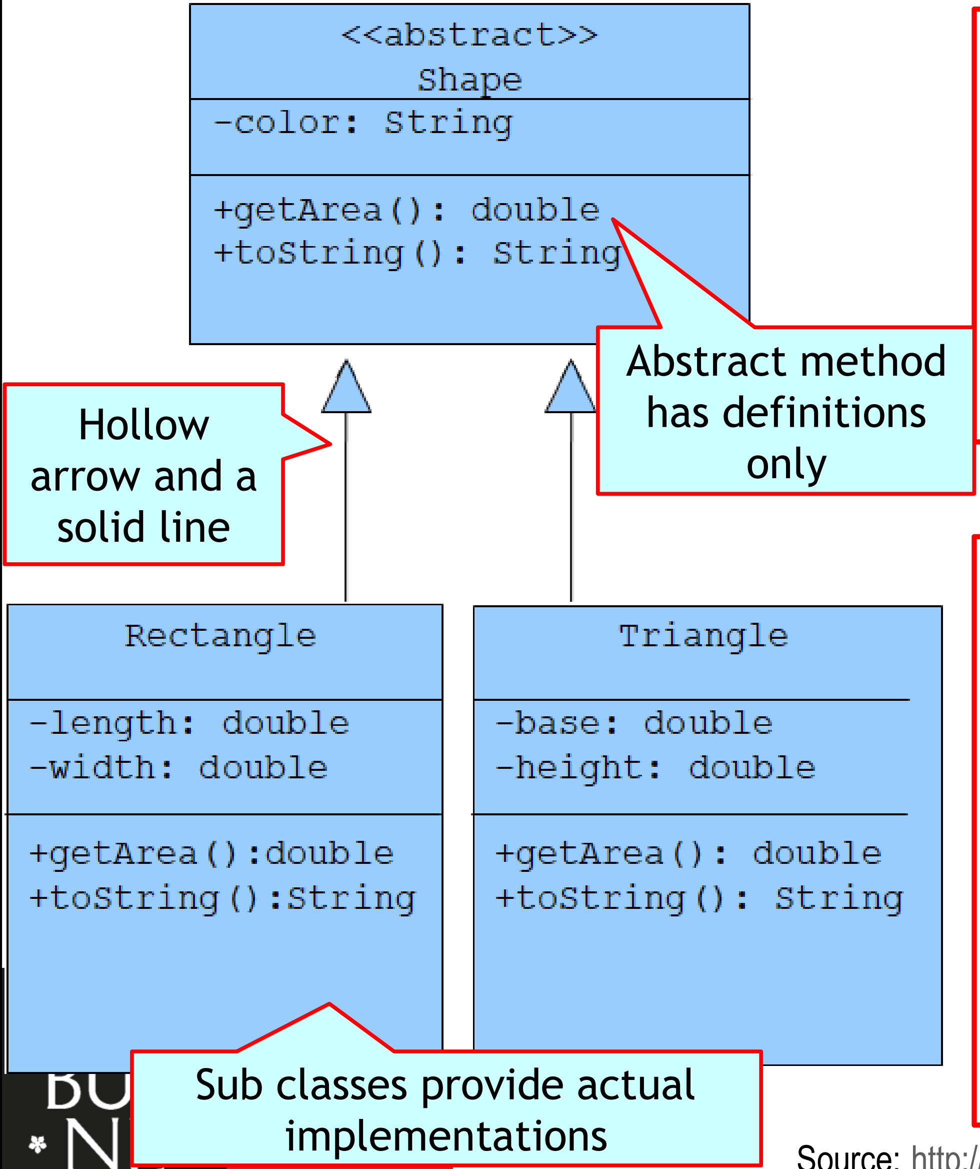
Abstract Methods

- Notice that the key word `abstract` appears in the header, and that the header ends with a semicolon.

```
public abstract void setValue(int value);
```

- Any class that contains an abstract method is automatically abstract.
- If a subclass fails to override an abstract method, a compiler error will result.
- Abstract methods are used to ensure that a subclass implements the method.

Abstract classes



<<abstract>>
Shape

-color: String

+getArea(): double
+toString(): String

Rectangle

-length: double
-width: double

+getArea():double
+toString():String

Triangle

-base: double
-height: double

+getArea(): double
+toString(): String

Hollow arrow and a solid line

Abstract method has definitions only

Sub classes provide actual implementations

```
public absract class Shape
{
   ….
   public abstract double getArea();
   ….
   ….
}
```

```
public class Rectangle extends
Shape
{
   ….
   public double getArea()
   {
          //method implementation
   }
   ….
   ….
}
```

Source: http://javaessential.com/java-tut/core-java_Encapsulation.php

Overriding

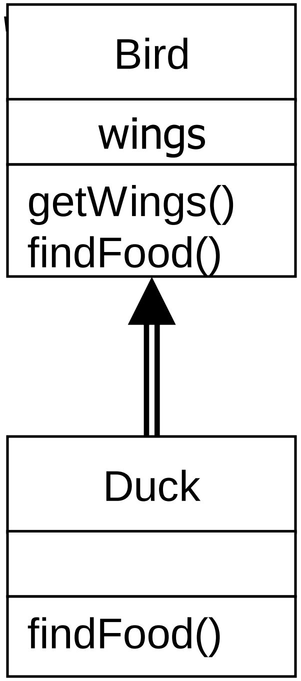
- Overriding a method is creating a method in a subclass that has the same signature (name, number, and type of arguments) as a method in a superclass.
  - That new method then hides the superclass's method.
  - In subclass can still call the superclass version by the prefix *super*

Example

```
class Bird
{  . . .
    public void findFood()   {
        System.out.println("I find food somehow!

    }
   . . .
}


class Duck extends Bird
{
    public void findFood()   {
        System.out.println("I look for insects on water!");
    }
}
```



Bird

wings

getWings()
findFood()

Duck

findFood()

Overriding

```
class Person {
String name;
public void greet ()  {
     System.out.println(“Good Morning”);
}
}
```

```
 public class Frenchman extends Person
    public void greet ()  {
       System.out.println(“Bon Jour”);
    }
}
```

The Frenchman class inherits name variable and greet function from Person class. It does not repeat declaration of name variable but repeats definition of greet function. A sub class may have it’s own definition of a function. This is called overriding.

Abstract methods and classes

- Method syntax:

  visibility **abstract** *type name* (*param_spec*) ;

- Class must be tagged as abstract:

  visibility **abstract** class *name* {

- Such a class is not instantiable

Example

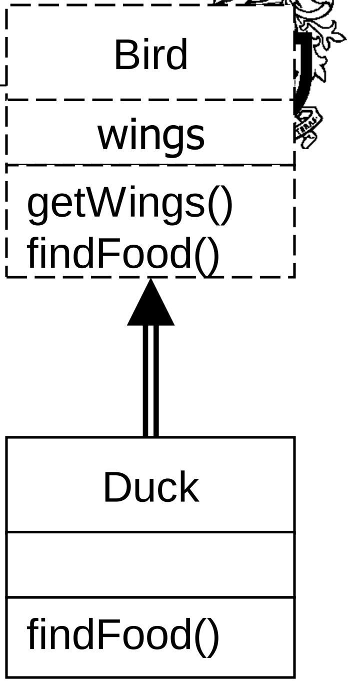
```
abstract class Bird
{  . . .
    public abstract void findFood();
   . . .

}


class Duck extends Bird
{
    public void findFood()   {
        System.out.println("I look for insects on water!");

    }

}
```



□concrete class vs abstract class

What's legal?

```
Duck daffy = new Duck();

System.out.println("daffy has " + daffy.getWings() + " wings");

daffy.findFood();



Bird dodo = new Bird(); // syntax error
```

Choosing overriding or abstract

- You can design a class with methods to be overridden when some subclasses would like to keep the method as such but some other subclasses would modify the method

- Designing a class with abstract method would be done when it would not make sense for the super class method to define itself in a concrete way

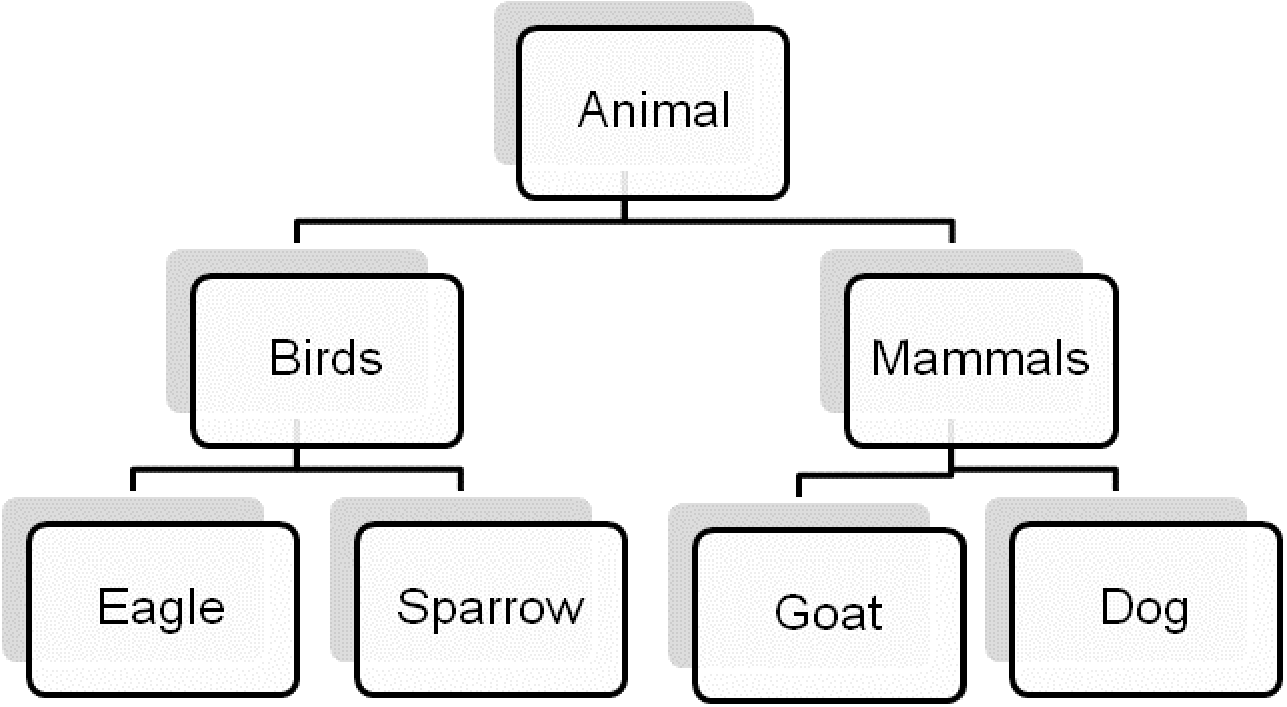
Animal class

To understand abstract methods, let’s take an example of the Animal class

The Animal class could comprise of very basic features of animals which are common to all animals

This class could be later on extended by classes like birds, mammals and fish etc. Which could add or modify their own features and behaviours

Animal class hierarchy



Extending from Animal class

```
public abstract class Animal  {
    public abstract void speak ();
}

class Dog extends Animal  {
  public void speak() {
     System.out.println("Ruf Ruf");
  }
}

class Goat extends Animal  {
   public void speak() {
       System.out.println("Baa Baa");
   }
}
```

Since all animals speak, the function have been included in the super class Animal but without an implementation. You can’t describe how an animal speaks because each animal speaks differently but all animals do speak. So we added a function speak in the Animal class as abstract.

Each animal sub-class will have its own different implementation of speak function as demonstrated here by the Dog and Goat class.

SWINBURNE UNIVERSITY OF TECHNOLOGY

- What is an interface?

What is an interface?

- Looks like a class but
  - All methods do not have implementations (no method body)
  - All methods are public
    - Methods can be abstract, default or static
  - There are no constructors.
  - Can contain fields which are public, static and final (they are called constants)

```
public static final int JANUARY = 1;
```

  - You can never instantiate an object from an interface.

An interface describes a set of methods that can be called on an object, but does *not* provide concrete implementations for all the methods.

SWINBURNE UNIVERSITY OF TECHNOLOGY

Where do interfaces come from?

- Programmers write interfaces
  - □ To specify common functionality to a class or number of unrelated classes

```
public interface Bicycle
{
    // wheel revolutions per minute
    public void changeCadence(int newValue);
    public void changeGear(int newValue);
    public void speedUp(int increment);
    public void applyBrakes(int decrement);
}
```

A bicycle's behaviour specified in an interface

A group of related methods with empty bodies

Source: https://docs.oracle.com/javase/tutorial/java/concepts/interface.html

Interfaces

- The general format of an interface definition:

```
public interface InterfaceName
{
   (Method headers...)
}
```

- We create an interface to specify behavior for other classes.
- By default, all methods in an interface are public.

Interfaces

- If a class implements an interface, it uses the `implements` keyword in the class header.

```
public class FinalExam3 implements Relatable
```

- An interface can contain field declarations:
  - □ However, all fields in an interface are `final` and `static`.
- You must provide an initial value as they are `final`, .

```
public interface Doable
{
  int FIELD1 = 1, FIELD2 = 2;
  (Method headers...)
}
```

Implementing Multiple Interfaces

- A class can be derived from only one superclass.
- Java allows a class to implement multiple interfaces.
- When a class implements multiple interfaces, it must provide the methods specified by all of them.
- In a class definition, after the implements key word, specify all the names of the interfaces to be implemented as below.

```
public class MyClass implements Interface1,
                                Interface2,
                                Interface3
```

What does it mean to implement an interface?

- Classes that implement the interface
  - Will have the behavior defined by the interface
- When you Implement the Interface you must supply an implementation for each method defined by the Interface
- Your class will not compile unless it has implemented all the required methods

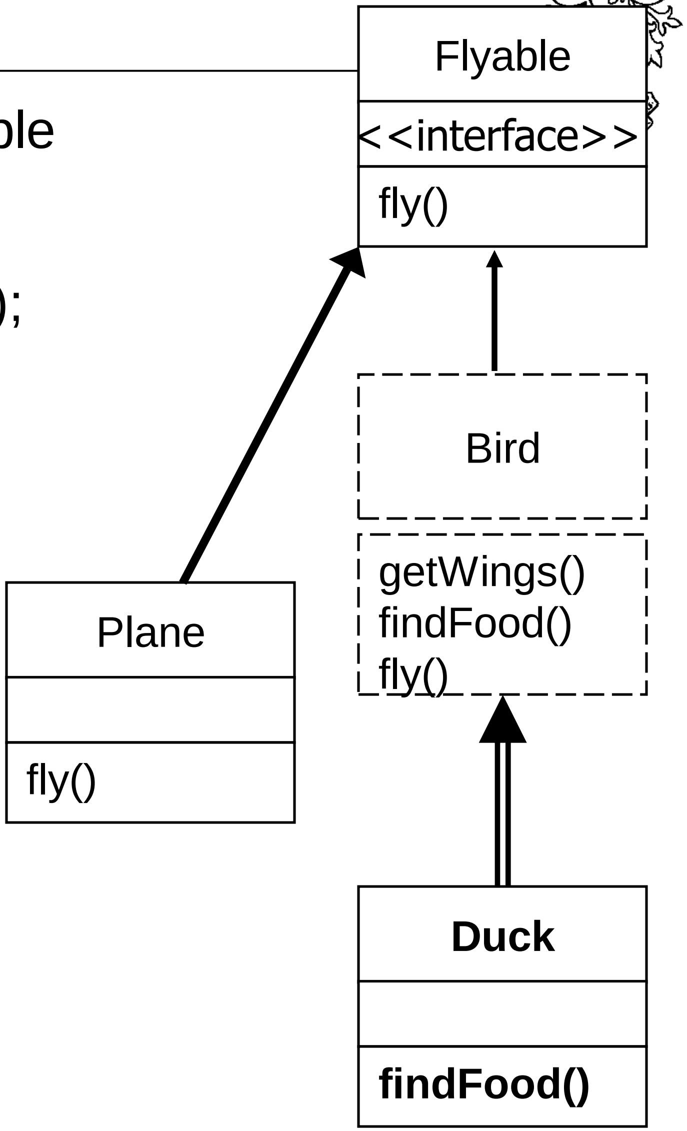
Interface

```
public interface Flyable
{
    public void fly();
}
```

□ That means that any class that implements the Flyable interface is required to have method fly() which is void and doesn’t take any parameters.

Example of an Interface

- abstract class Bird **implements** Flyable
  ```
  { . . .
     public void fly() {
         System.out.println("Flap! Flap!");
    }   . . .
  ```
- }
- class Duck extends Bird . . .
- class Plane **implements** Flyable . . .
  ```
  {
     public void fly() {
         System.out.println("Brmmm!");
    }
    . . .
  ```



- Why would you use an Interface?

Why would you implement an Interface

- “There are a number of situations in software engineering when it is important for disparate groups of programmers to agree to a "contract" that spells out how their software interacts.”
- “Each group should be able to write their code without any knowledge of how the other group's code is written. Generally speaking, *interfaces* are such contracts.”

An interface defines a contract.
A class or structure that implements an interface must adhere to its contract.

Source: https://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html

Why would you implement an Interface

- Improves design,
  - □ Encourages Strong separation of functionality from implementation.
  - □ Clients interact independently of the implementation.

- What does it mean to implement an interface?

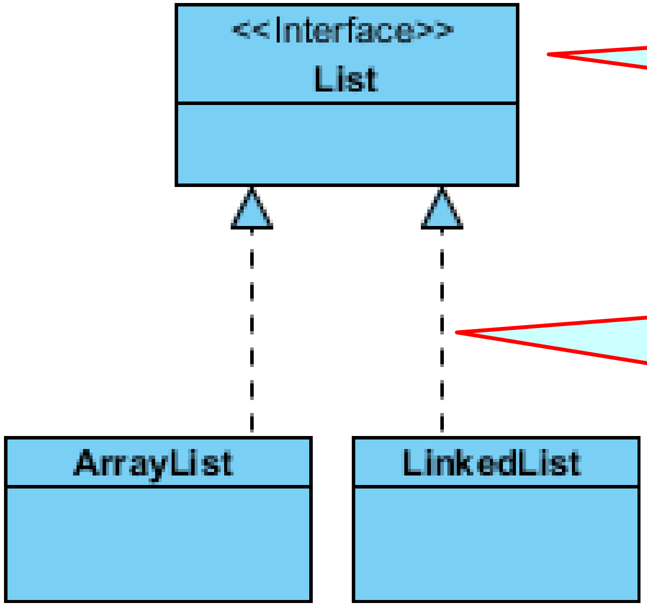
What does it mean to implement an interface?

- Classes that implement the interface
  - Will have the behaviour defined by the interface
- When you Implement the Interface you must supply an implementation for each method defined by the Interface
- Your class will not compile unless it has implemented all the required methods

- How do we show an Interface in a class Diagram?

Showing an Interface in a Class Diagram

- A class is telling the world, I will have the behaviour defined by the Interface.



Annotate with <<interface>>

Hollow arrow and a dashed line showing the relationship between the class and interface

How do we implement an interface?

□Select the Interface

□Change class definition

□Add defined methods

□Implement the methods

□Test it

Step 1 – Select the interface

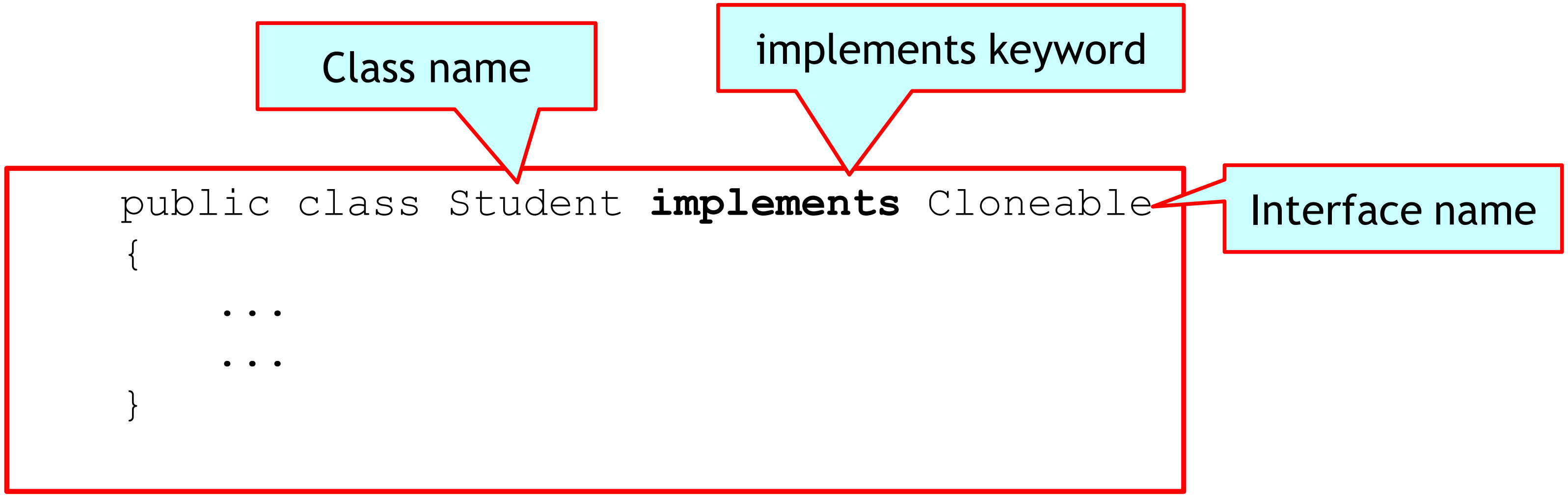
- Define or find the Interface you wish to implement
- Make sure you understand what it does

Example: A class that implements this interface will be able to return a clone (a field to field copy) of itself when the clone method is called.

```
interface Cloneable
{
    public abstract Object clone();
}
```

Step 2 – Change the class definition

- Show that the class will implement the interface



Class name

implements keyword

Interface name

```
public class Student implements Cloneable
{
    ...
    ...
}
```

Step 3: Add methods defined in the interface

- Add the method headers for each required method that this class will implement

```
public class Student implements Cloneable
{
          public Object clone()
          {

          }
}
```

Add the method specified in the interface

Step 4: Implement the methods

- Write the method bodies

```
public class Student implements Cloneable
{
       public Object clone()
       {
             Student s = new Student();
             s.name = this.name;
             //Set other fields here
             ...
             ...
             return s;
       }
}
```

Implement method body

- If this class does not need to implement all the interface’s methods, these unimplemented methods still need to be presented in the class but marked as “abstract”.
- In this case, the class becomes an “abstract” class.

SWINBURNE UNIVERSITY OF TECHNOLOGY

Using an interface as a Type

- When you define a interface, you are defining a new reference data type.
- You can use interface names anywhere you can use any other data type name.
- If you define a reference variable whose type is an interface, any object you assign to it *must* be an instance of a class that implements the interface.

```
interface T1
{
        ...
}
public class A implements T1
{
        ...
}
```

```
A a = new A();
boolean test = (a instanceof T1);
```

This results true because *a* is a type of *T1*. Therefore, T1 t = (T1) a; can be done

Step 5: Test

- TEST your work
  - □ Declare a variable of the interface type

```
Cloneable  i;
```

  - □ Assign an object

```
i = new Student();
```

  - □ Test the methods

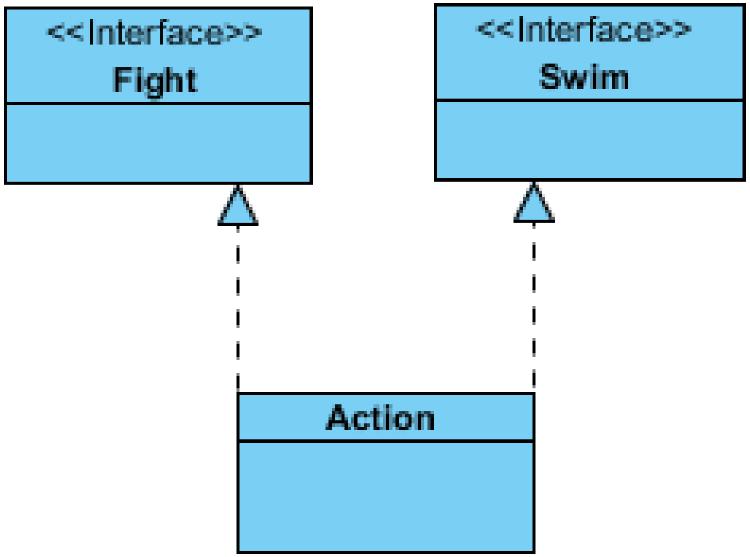
```
Student other = (Student)i.clone();
```

  - □ Try instance of

```
if (other instanceof Cloneable)
{
        ...
}
```

Implementing many interfaces

- A class can implement many interfaces



```
interface Fight
{
        ...
}
interface Swim
{
        ...
}
public class Action implements Fight, Swim
{
        ...
}
```

This is not multiple inheritance!
Class Action is agreeing to implement the behaviour of Fight and Swim...

Inconsistency issues when implementing multiple interfaces

- It is possible that two interfaces have two inconsistent methods. For example, the following two interfaces have the same method signature, but different return types.

```
public class AClass
implements A, B
{
   public int getY() {…}
   public double getY() {…}
}
```

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

```
public interface B
{
  public double getY();
}
```

We will have problems here, because Java does not allow overloading based on the return type. (Overloading happens with different parameters)

- Compiler will give error message for AClass – “getY() is already defined”
  - □ Programmers have to check the inconsistency before choosing interfaces

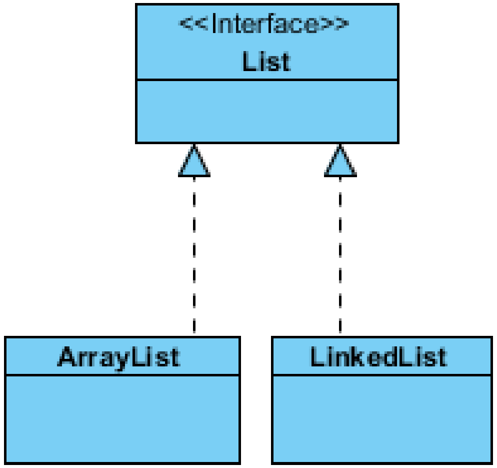
- So where's the polymorphism?

Polymorphism with Interfaces

- Java allows you to create reference variables of an interface type.
- An interface reference variable can reference any object that implements that interface, regardless of its class type.
- This is another example of polymorphism.
- Example:
  - RetailItem.java
  - CompactDisc.java
  - DvdMovie.java
  - PolymorphicInterfaceDemo.java

Interfaces support polymorphism

- Many different objects can implement the same interface
- They outwardly present the same methods
- But perform the job in their own way.



Cloneable interface example…

- Student, Lecturer, Donkey could all implement Cloneable, then;
  - Each one of these would then have a clone method
  - Each one of these would be seen to be Cloneable
  - Each one could be added to a collection of Cloneable
  - Each one will clone in its own way

Interfaces vs Inheritance

- ■ Inheritance implements the “is a” relationship for groups of classes related by **classification**.
  - □ Student is a kind of Person
  - □ Lecturer is a kind of Person
- ■ Interfaces implement the “is a” relationship in terms of **functionality**. All classes that implement an interface have the functionality of the interface.
  - □ If **Person, Cost and Account** classes are implementing the **Comparable** interface in Java
    - □ Person is **Comparable** with another Person
    - □ Cost is **Comparable** with another Cost
    - □ Account is **Comparable** with another Account

Interface and Inheritance together?

So far...

```
class ACMEBicyle implements Bicycle
{
           ...
}
```

I implement Bicycle behaviour

```
public class Hunter implements Actor, Drawable
{
    ...
}
```

I have the behaviour defined by the Actor and the Drawable interfaces

This means Fox is a Animal and has behaviour specified by Drawable interface (can be drawn)

```
public class Fox extends Animal implements Drawable
{
    ...
}
```

Any class CAN **ONLY** extend **one superclass**, but CAN implement **more than one interface**

Extending Interfaces

- An interface can extend another interface in the same way a class does.

```
interface Vehicle
{
        //move back and forward
}
interface Airplane extends Vehicle
{
        //move up and down
}
```

You are not actually inheriting but accumulating behaviour into one interface…

- Since interfaces are not classes, interfaces can extend multiple interfaces.

Some Commonly Used Interfaces in API

- Compar***able<T>***
- Comparator<T>
- Serializ***able***
- Clone***able***

Using Comparable<T>

- Objects to be compared need to have their classes implementing the Comparable<T> interface
  - Comparable<T> has only one method compareTo()
- "Lists (and arrays) of objects that implement Comparable can be sorted automatically by Collections.sort() (and Arrays.sort()).
- **Comparable in Java** is used to implement natural ordering of objects

Source: https://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html

Comparable<T> Example 1

```
public class Employee implements Comparable<Employee> {
    private int empID;
    private String ename;
    private double sal;
    private static int i;
    public Employee() {
        empID = i++;
        ename = "dont know";
        sal = 0.0;
    }
    public Employee(String ename, double sal) {
        this.empID=i++;
        this.ename=ename;
        this.sal=sal;
    }
    public String toString() {
        return “empID "+empID+"\n ename "+ename+"\n sal "+sal;
    }
    public int compareTo (Employee o1) {
        if(this.sal == o1.sal)
            return 0;
        else if (this.sal > o1.sal)
            return 1;
        else return -1;
    }
}
```

This will allow the sorting in ascending order based on the value of “sal”

Comparable<T> Example 1 cont…

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;
import java.util.List;
public class ComparableDemo {
    public static void main(String[] args) {
        List<Employee> ts1=new ArrayList<Employee>();
        ts1.add(new Employee ("Tom",40000.00));
        ts1.add(new Employee ("Harry",20000.00));
        ts1.add(new Employee ("Maggie",50000.00));
        ts1.add(new Employee ("Chris",70000.00));
        Collections.sort(ts1);
        Iterator itr =ts1.iterator();
        while(itr.hasNext()) {
            Object element = itr.next();
            System.out.println(element + "\n");
        }
    }
}
```

Output:

```
empID 1
ename Harry
sal 20000.0
empID 0
ename Tom
sal 40000.0
empID 2
ename Maggie
sal 50000.0
empID 3
ename Chris
sal 70000.0
```

Collections.sort() uses the compareTo method to know how to compare pairs of items in the list.

In the compareTo():

- `if(this.sal == o1.sal)` returns 0 to indicate the two values are equal
- `if (this.sal > o1.sal)` returns 1 to indicate first sal is larger than the other
- `else` returns -1 to indicate first sal is smaller than the other

`equals`() and `compareTo`()

- When overriding equals() and implementing compareTo() at the same time, it is recommended to ensure the following:
  - □ If a.equals(b) returns `true`, a.compareTo(b) must return 0
  - □ If a.equals(b) returns `false`, a.compareTo(b) must return non-zero

Using Comparator<T>

- Is in the java.util package
- The comparator specifies two methods to implement
  - □ public int compare(Object o1, Object o2)
    - □ Compares its two arguments for order. Returns a negative integer, zero, or a positive integer if the first argument is less than, equal to, or greater than the second.
  - □ public boolean equals(Object obj)
    - □ Indicates whether some other comparator object is "equal to" this Comparator.

Comparator<T>

```
import java.util.Arrays;
public class Demo
{
   public static void main(String[] argv) throws Exception
   {
        String strs[] ={"d","h","a","c","t"};
        MyComparator rsc = new MyComparator();

        Arrays.sort(strs, rsc);
        for (String s : strs)  System.out.println(s + " ");
        Arrays.sort(strs);
        System.out.println("Sorted in asc order: ");
        for(String s : strs)   System.out.println(s + " ");
   }
}
```

Comparators can be passed to a sort method (such as Collections.sort or Arrays.sort) to allow precise control over the sort order.

```
import java.util.Arrays;
import java.util.Comparator;
class MyComparator implements Comparator<String>
{
    public int compare(String strA,String strB)
    {
        return strA.compareTo(strB);
    }
    ….
}
```

Interested readers can study the API

Comparator<T>

```
import java.util.Arrays;
import java.util.Comparator;
Public class EmpComparator implements Comparator<Employee>
{
    public int compare(Employee emp1,Employee emp2)
    {
        //compare salary first
        int diff = emp1.getSalary() - emp2.getSalary();
        if(diff == 0) //if salary is the same compare name
        {
                diff = emp1.getName().compareTo(emp2.getName());
        }
        return diff;
    }
}
```

Comparators can be passed to a sort method (such as Collections.sort or Arrays.sort) to allow precise control over the sort order.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;
import java.util.List;
public class ComparatorDemo {
    public static void main(String[] args) {
        List<Employee> ts1=new ArrayList<Employee>();
        ts1.add(new Employee ("Tom",40000.00));
        ts1.add(new Employee ("Harry",20000.00));
        ts1.add(new Employee ("Maggie",50000.00));
        ts1.add(new Employee ("Chris",70000.00));
        Collections.sort(ts1, new EmpComparator());
        …
        …
    }
}
```

More examples can be found from the Internet

e.g.

https://www.geeksforgeeks.org/collections-sort-java-examples/

Some thoughts on when to use Comparator and Comparable…

- When developing the class, if you know how it should be sorted and/ or it is the natural way of sorting then use Comparable.
- But if an object can be sorted in multiple ways and during the development of the class this is not clear or if the client is specifying on which parameter sorting should take place then use Comparator interface.
- Also if you don’t have access to the class to implement the Comparable interface than use a separate class which implements Comparator interface.

References

- https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html
- https://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html
- How to Program Java, 9th Edition, Deital and Deital, Chapter 10
- Chapter 10 - Objects first with Java – Barnes & Kolling