COMP122 Week 7

Abstract Classes and Interfaces

Dr. Patrick Totzke
totzke@liverpool.ac.uk

`https://liverpool.instructure.com/courses/59716`

This Week

Today: more on inheritance in Java

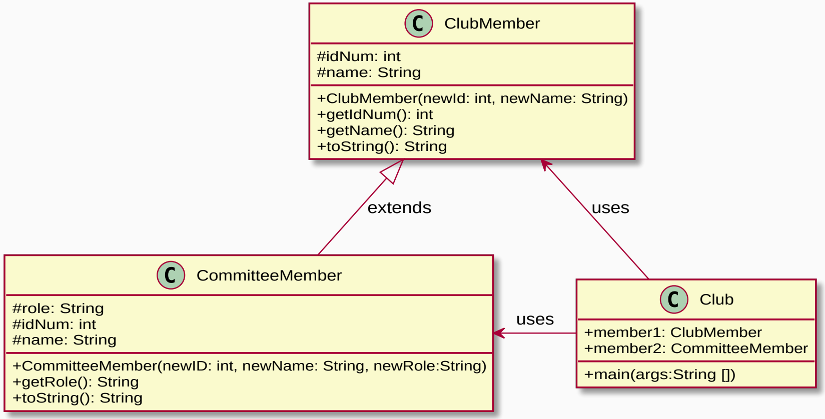
- Arrays
- Overriding methods
- A closer look at Constructors and the keyword `super`

Thursday

- Abstract classes
- Interfaces
- Assignment 2

Back to the Club example

Club Example – Full UML Class Diagram



Implementation of ClubMember

```
public class ClubMember {

     /* Attributes */
    protected int idNum;
    protected String name;

    /* Constructor */
    public ClubMember(int newID, String newName) {
        idNum = newID;
        name = newName;
    }

    /*  Other methods */
    public int getIdNum() { return idNum; }
    public String getName() { return name; }

    public String toString() {
        return("Member Name: " + getName() + " ID number: " + getIdNum());
    }
}
```

Implementation of CommitteeMember

```java
public class CommitteeMember extends ClubMember {

      /* additional attribute */
    protected String role;

    /*  Constructor */
    public CommitteeMember(int newID, String newName, String newRole) {
        super(newID, newName);
        role = newRole;
    }

    /*  getter for new atribute */
    public String getRole() { return role; }

    /* override toString method*/
    public String toString() {
        return("Committee Member Name: " + getName() +
                   "  ID number: " + getIdNum() + " Role: " + getRole());
    }
}
```

Club Example – An Application

```java
/** A short application to test ClubMember and CommitteeMember  */
public class Club {

  public static void main(String[] args) {

    ClubMember member1 = new ClubMember(123, "Fred");
    CommitteeMember member2 = new CommitteeMember(22, "Helen", "Chair");

    System.out.println(member1.toString());
    System.out.println(member2.toString());
  }
}
```

```
$> java Club
Member Name: Fred  ID number: 123
Committee Member Name: Helen  ID number: 22 Role: Chair
```

Interlude on Arrays

Arrays

Beyond primitive data types, Java has array data structure which allows to store fixed-length lists of values with the same type.

The length of the array must be specified when the array is declared.

Examples of array declaration statements include:

```
long[] numbers = new long[10];
double[] reals = new double[200];
Person[] rectangleSet = new Person[15];
boolean[] truthValues = new boolean[18];
```

Remarks about arrays

- Note the square brackets in the declaration statement. The first set of brackets informs the Java compiler that you are declaring an array. The second set of brackets denotes the size of the array inside the brackets.
- For any type `myType` there is a corresponding array type `myType[]`,
- Declaring an array uses the `new` keyword.
- The size of the array can itself be read from a variable:

```
int size = 250;
double[] reals = new reals[size];
```

Remarks about arrays (cont.)

- An array has a "read-only" (technically, a `final int`) attribute called `length`, which tells you the size of the array (the declared size).

```java
size = reals.length;
```

- An array element is accessed using square bracket notation.

```java
x = reals[12];
y = reals[13] * reals[10];
```

- Java arrays are 0-indexed, i.e. the first element has index 0, and the last element in the array has index $length - 1$.

Remarks about arrays (cont.)

- Arrays of primitive data types will be initialized with “sensible” values (e.g. 0 for arrays of `int` values, `false` for arrays of `boolean` values, etc.).
- Attempting to access an element that is outside the index limits will result in an `ArrayIndexOutOfBoundsException` that will terminate your program (unless you are checking for that exception, see week 9).

An Example that uses Arrays and Polymorphism

```
public class ArrayClub {
  public static void main (String [] args){
    // Let's declare an Array of Members
    ClubMember[] members = new ClubMember[3];

    // and fill it with different members
    members[0] = new ClubMember(123, "Fred");
    members[1] = new ClubMember(124, "Olaf");
    members[2] = new CommitteeMember(22, "Helen", "Chair");

    int i;
    /* Arrays have a (read-only) attribute called "length" which is
     * convenient when iterating over their elements */
    for (i = 0; i < members.length; i++) {
      System.out.println(members[i].toString());
    }
  }
}
```

Overiding methods

Overriding a Superclass Method

Given a class hierarchy, a subclass will inherit the `public` or `protected` attributes and methods in the superclass.

However, if the subclass includes a public or protected method with the same ”signature” (i.e. the same name and parameter list) as a method in the superclass, the superclass method will not be inherited.

We say that the subclass method overrides the superclass method.

Note the accessibility of the subclass method must be the same or stronger than that in the superclass or you get a syntax error.

Overriding vs Overloading

Note that overriding should not be confused with overloading.

Overriding involves providing several protected/public methods with the same name and the same parameter list, but declared in classes which are in a subclass/superclass relationship.

Overloading involves providing several methods with the same name, but having different parameter lists (e.g. `println` methods). For example, often several constructors might be defined for a class, with different numbers of parameters provided to them (say, where other parameters are assigned “default” values in the constructor).

Overriding I

A subclass may replace an inherited method by giving a new definition. The superclass method is said to have been *overridden*.

Example
Add to `CommitteeMember` the following method:

```
        // method
        public String getName() {
            return ("Committee Member: " + name);
        }
```

Recall we declared `member2` as follows.

```
CommitteeMember member2 = new CommitteeMember(22, "Helen", "Chair");
```

What happens when `member2.getName()` is called and the output printed in the `main` method?

Overriding II

Previously (without overriding `getName`) we obtained

```
Helen
```

now we obtain the following.

```
Committee Member: Helen
```

The `getName()` method from `ClubMember` has been overridden by the `getName()` method from `CommitteeMember`.

The `toString()` method is another example of overriding a class method. It is defined in both classes, and has the same signature (no parameters in this case).

java.lang.Object

The most general class in Java's object library is `Object`.
(see `https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html`)

It is a superclass of all other classes, the root of any class hierarchy. If a class does not explicitly state that it extends another class it is assumed to be a subclass of `Object`.

All subclasses of `Object` inherit the public and protected methods in `Object`.

One of them is the `toString()` method we have already seen.

Sidenote on the toString() method

The `toString()` method can be a useful for debugging purposes.

Strictly speaking, you are overriding this method from the class `Object`. One side-effect of this is that any object `obj` is automatically cast into `String`'s where necessary:

We could replace

```java
System.out.println(obj.toString());
```

with this line

```java
System.out.println(obj);
```

(The `toString()` method is automatically called in this case.)

Inheritance and Constructors

Inheritance and Constructors

A subclass does not inherit constructors from its superclass!

However, we can call the superclass constructor using the keyword `super`, as in

```
super(newID, newName);
```

If a subclass constructor does not explicitly invoke a superclass constructor, Java will automatically invoke the default superclass constructor i.e. the one with no parameters. It does this before executing the code in its own constructor.

If one of the superclasses has no default constructor, this will cause a syntax error.

Inheritance and Constructors – Example

```java
class A {
    public A () { System.out.print("A1"); }
    public A ( int i ) { System.out.print("A" + i ); }
}
class B extends A {
    public B () {
        super(2);
        System.out.print("B");
    }
}
class C extends B {
    public C () { System.out.print("C"); }
}

class TestSuper {
    public static void main (String[] args) {
        C obj = new C ();
    }
}
```

Output

The previous program will output the following.

```
$ java TestSuper
A2BC
```

If we comment out the line 7 we will obtain

```
$ java TestSuper
A1BC
```

If we comment out both 7 and 2 we will obtain a syntax error

```
TestSuper.java:6: error: constructor A in class A cannot be applied to given
    types;
    public B () {
                ^
  required: int
  found: no arguments
  reason: actual and formal argument lists differ in length
```

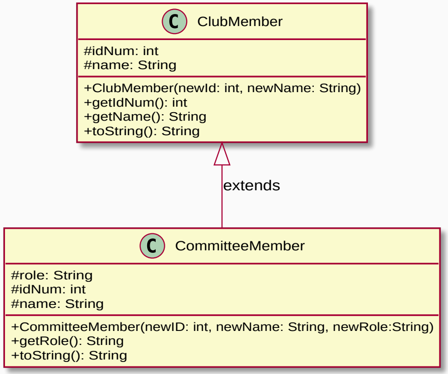
as Java tries to invoke A's default constructor and doesn't find one.

Abstract Classes

Last Week: Inheritance

We learned about Java's basic inheritance mechanisms

- the keywords `extends` and `super`
- Overloading
- Constructors



Recall: The Club Example

```java
public class CommitteeMember extends ClubMember {

  protected String role;

  //*  Constructor
  public CommitteeMember(int newID, String newName, String newRole) {
     super(newID, newName);
     role = newRole;
  }

  //*  Methods
  public String getRole() { return role; }

  public String toString() {
     return("Committee Member Name: " + getName() +
              "  ID number: " + getIdNum() + " Role: " + getRole());
  }
}
```

Abstract Methods and Classes

In constructing a hierarchy of classes, it is sometimes beneficial to include a top class whose methods *cannot* be instantiated and acts as a *blueprint* for all subclasses.

Example

Suppose you want to replicate the whole Clubmember/CommitteeMember hierarchy, but slightly differently. Say, membership in a university, or shopping club..

An *abstract method* is defined by the method's signature and has no method body.

An *abstract class* is a class that contains at least one abstract method.

Syntax for Abstract Classes and Methods

The Java keyword `abstract` is used to specify abstract classes.

```
modifiers abstract class ClassName { .... }
```

Abstract methods have no body, just the method signature

```
modifiers abstract type methodName ( parameters );
```

Remarks

- Java uses abstract classes a lot in graphical packages.
- Any class containing an abstract method must be declared abstract.
- Abstract classes cannot be instantiated. They must be subclassed.
- A subclass of an abstract class can be instantiated only if it implements all of the abstract methods of the superclass.
- If a subclass of an abstract class only implements some (but not all) of the superclass abstract methods it also must be declared as abstract.

Abstract Classes: Example

Going back to our ClubMember/CommitteeMember example, we could create an abstract class called `Member` which is a superclass of `ClubMember`.

We use `public abstract class Member` { ... } to do this.

This has two abstract methods `getName()` and `getIdNo()` which just contains the method's signature (but no body).

Then every (non abstract) subclass *must* implement `getName()` and `getIdNo()`.

Abstract Classes: Member

```java
/** Member, an illustration of an abstract class. **/

public abstract class Member {

    //  Abstract method getName() meaning subclasses
    //  must implement this method (or themselves be abstract).
    public abstract String getName();

    //  Abstract method getIdNum() meaning subclasses
    //  must implement this method (or themselves be abstract).
    public abstract int getIdNum();
}
```

Note that the abstract methods `getName()` and `getIdNo()` contain their signature (e.g. visibility, name and return type), but no body, not even { }.

Abstract Classes: ClubMember

The `ClubMember` class is as it was before but we make it a subclass of `Member`.

```java
/**  ClubMember class file, an extension of the abstract Member class. */
public class ClubMember extends Member {
     ..................
     ..................

     //*  Implementation of abstract methods
     public int getIdNum() {
          return idNum;
     }
     public String getName() {
          return name;
     }
      ................

}
```

Polymorphism

Continuing with our previous example, with abstract `Member` class, extended by `ClubMember` and further by `CommitteeMember`.

```java
ClubMember member1 = new ClubMember(123, "Fred");
CommitteeMember member2 = new CommitteeMember(22, "Helen", "Chair");
```

We can instead assign the same objects to a variable of type `Member`, the superclass.

```java
Member member1 = new ClubMember(123, "Fred");
Member member2 = new CommitteeMember(22, "Helen", "Chair");
```

When we call `member1.getName()` since "Member" has the abstract method `getName()`, Java will look in the subclasses of "Member" to find a suitable implementation of `getName()`.

Polymorphism (cont.)

This is also true of the method `toString()`. The exact version of this method that is called will depend upon the instance of the object in question.

```java
public class TestAbstract {
    public static void main(String[] args) {
        Member member1 = new ClubMember(123, "Fred");
        Member member2 = new CommitteeMember(222, "Helen", "Chair");
        System.out.println(member1.toString());
        System.out.println(member2.toString());
    }
}
```

```
Club Member Name: Fred  ID number: 123
Committee Member Name: Helen  ID number: 222 Role: Chair
```

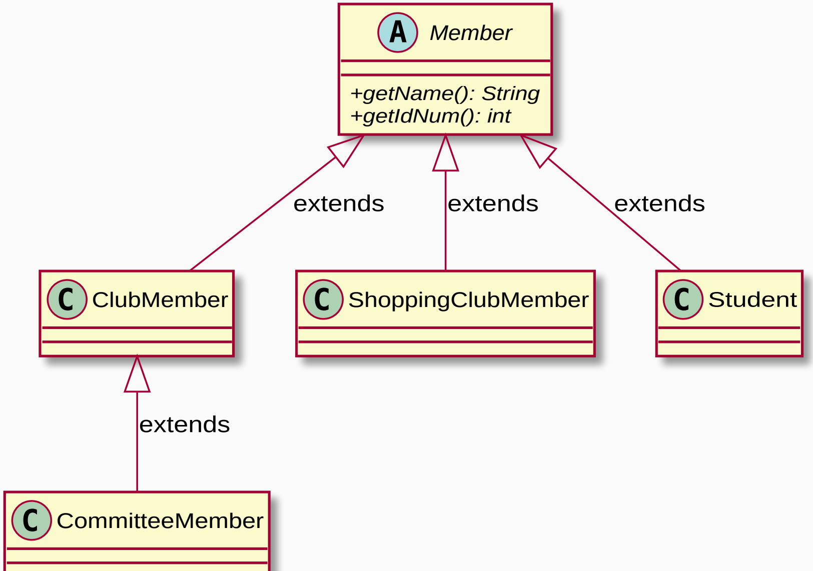
Another subclass of “Member”

Another advantage is we can define new subclasses of `Member` without having to re-define or re-compile the rest of the classes in the existing (Club) hierarchy.

I could, for example, define a new subclass `ShoppingClubMember` if I were interested in creating an application dealing with people who shop at (say) Costco. Or maybe I can create the subclass `Student` to deal with university students (or any other school).

In each case, the person in question will have a name (which might actually consist of several parts/instance variables), and a number to identify them in some manner. In each case it will be useful to have methods to get their names (in a form usable to the application) and their “idNum” that we use to help identify them.

A possible class hierarchy



Abstract Classes: Why?

By declaring a class `abstract` you can

1. prevent it from being instantiated
2. enforce that concrete (instantiable) subclasses override some method.
3. provide concrete methods with “holes” – black boxes provided by concrete subclasses.

Polymorphism

In Java, polymorphism essentially refers to three different situations:

- overriding an inherited method;
- implementing an abstract method;
- implementing an interface (next up).

Interfaces

Multiple Inheritance

Many, if not most, Object oriented programming languages allow for some form of multiple inheritance, i.e. a class being a subclass of several different super-classes, inheriting all attributes and methods of each of the parent classes. For example, Python and C++ allow this directly.

Java does not allow this (for ordinary classes), but for interfaces.

Interfaces

An **interface** is a (special) class that contains only abstract methods and/or constants.

Thus, an interface supplies a specification of methods which must be implemented by a subclass of the interface. In that case, we say that the subclass **implements** the interface.

Interfaces – an Example

The Java keyword to use is `interface`, and this is used in place of `class` when declaring the interface.

```java
public interface Transformable  {

        final static int MAX_ITERATIONS = 10;

        public void alterView();
        public void recolor(int newColor);
        public double computeArea(double sideLength, double angle);
}
```

These declarations define the "signature" of the method, i.e. what parameters (and their types) they take, and what the return value of the methods are.

Interfaces (cont.)

Methods don't have to be declared `abstract`, since an interface is *automatically* an abstraction. All methods declared in an interface are `public` and `abstract` by default.

All attributes in an interface are automatically `public`, `static`, and `final`. Hence, these are really constants (i.e. trying to change them in a class that implements the interface will result in a compilation error).

Typically, interfaces identifiers end in "-able" by convention. Examples are `java.lang.Comparable` and `java.io.Serializable`.

Implementing an interface

We can define a class that implements an interface, or we can define a subclass that extends another class and also implements an interface.

```
public class NewClass implements Transformable {

    ...

     /* Must provide implementations of the methods declared
      * in the interface! */
}
```

Implementing an interface (cont.)

A (concrete) class can also implement several interfaces if desired. Such a class would have to implement all methods from all the interfaces it implements.

```java
public class B extends A implements Tranformable, Killable {
        ...
        /* methods from Transformable */
        /* methods from Killable */
}
```

Since an interface is itself a (special) class, it can `extend` (one or more!) other interfaces, but cannot `implement` them.

```java
public interface X extends Tranformable, Killable {
        ...
}
```

A concrete example (I)

The Bird class defines a (base) class for an object hierarchy.

```
public class Bird {
    protected String name;

    public Bird(String myName){
        name = myName;
    }

    public String getName() {
        return name;
    }

    public boolean flies() {
        return true;
    }

    public String type() {
        return "Bird";
```

A concrete example (II)

```
    }

    public void printDetails() {
        System.out.println(type() + " " + getName()
                                    + " flies: " + flies());
    }
}
```

The method `flies()` returns `true`/`false` depending upon whether or not the type of bird flies. By default it returns `true`, but we can override it in classes that inherit from `Bird`.

Making a bird fly

Suppose we want to supply some methods to represent the action of flying. For the purposes of illustration, we'll use an interface to do this.

What will we need? Flying (typically) consists of three types of behaviour: (1) Getting off the ground, (2) moving forward, and (3) landing again.

```java
public interface Fly {

    public void goUp();
    public void goForward();
    public void goDown();

}
```

Making a bird fly (I)

```
public class Bird implements Fly {
    protected String name;

    public Bird(String myName){
        name = myName;
    }

    public String getName() { return name; }
    public boolean flies() { return true; }
    public String type() { return "Bird"; }
    public String toString()  { return (type() + " " + name); }

    public void printDetails() {
        System.out.println(type() + " " + name + " flies: " + flies());
    }

    //**  Interface methods, defaults for a bird that flies.
    public void goUp()  {
        System.out.println(type() + " " + name + " starts to fly.");
```

Making a bird fly (II)

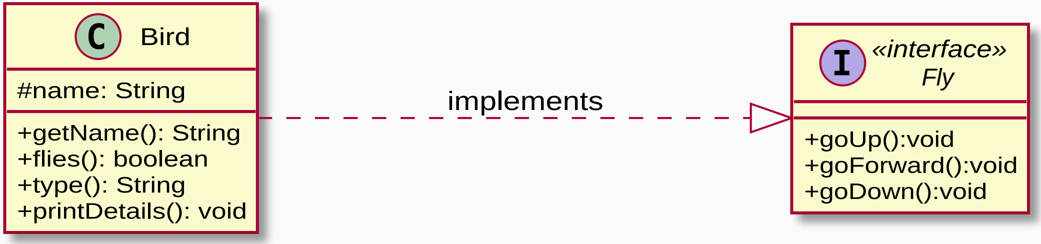
```java
    }

    public void goForward() {
        System.out.println(type() + " " + name + " flies forward.");
    }

    public void goDown() {
        System.out.println(type() + " " + name + " lands after flying.");
    }
}
```

UML diagrams for interfaces

We show an implementation of an interface in a UML diagram by using a dashed line from the class doing the implementation to the interface.



What about the Penguins?

As written, these three methods that implement `Fly` assume that a bird can fly. What if the bird can't fly?

Opt.1 We could rewrite these three methods, to check if a bird flies or not (by checking the `flies()` method) to accommodate flightless birds too (e.g. provide different output depending upon the output of the `flies()` method).

Opt.2 Alternatively, we could override the methods in a class that extends `Bird` to redefine these three methods.

Which alternative do you do? Whatever seems most "correct" to you. (There are arguments either way...)

Penguins don't fly... (I)

Let's go with Option 2 and define a Penguin class like this:

```java
public class Penguin extends Bird {

     // Constructor
     public Penguin(String myName) {
        super(myName);
     }

    // Overrides flies() as Penguins don't fly.
    public boolean flies() {
        return false;
    }

    public String type() {
        return "Penguin";
    }
    //  Override the methods about flying since a penguin can't.

```

Penguins don't fly... (II)

```java
    public void goUp() {
        System.out.println(name + " attempts to fly, but can't.");
    }

    public void goForward() {
        System.out.println(name + " wishes it could fly...");
    }

    public void goDown() {
        System.out.println(name + " makes a very small hop.");
    }
}
```

Note: We don't need to explicitly declare that `Penguin` implements `Fly` because the superclass `Bird` already does. But we can still override the methods.

Why use interfaces?

Interfaces can present a common set of methods that must be implemented by different classes.

Example: you may later write a `Plane`, `Helicopter`, or `Butterfly` class that implements the `Fly` interface...

Interfaces are used a lot in various Java libraries, such as ones providing graphics. These methods then must be implemented by these graphics objects and guarantee some common behaviour across all the objects, such methods for drawing themselves on screen, or re-sizing them, etc.

Interfaces define Datatypes

Just like abstract classes, you cannot instantiate interfaces directly:

```java
Fly animal = new Fly();  // ERROR!
```

... you can only instantiate classes that implement them ...

```java
Fly penguin1 = new Penguin("hugo");  // perfectly fine
```

In Java, an interface is a specification which public methods must exist in a class that *implements* it.

- Interfaces define a type,
- (typically) only contain constants and method signatures,
- cannot be instantiated,
- can be implemented by a class, which then has to contain all method bodies declared in the interface,
- can be extended by other interfaces.
- A class can implement more than one interface.

Interfaces v Abstract classes

Abstract Classes	Interfaces
can only `extend` one superclass	multiple inheritance between interfaces
can extend any class	can only extend interfaces
may have `abstract` and concrete methods	only abstract methods [1]
`protected` methods allowed	methods are `public abstract`
no restriction on attribute modifiers	only `public static final` vars

[1] not strictly true any more as of Java 8; check out the `default` keyword for added messiness

A2

Your Questions

Q38

Q: For part 2 of A2, are we expected to use HashMap?

A: I do not expect you to use HashMaps here as I have not yet introduced Collections.

If you want to use them, be my guest.

Q39

```
For MonoAlphaSubstitution, all examples for substitution strings

- Are no longer than 26 characters
- Are always an even number of characters
- Do not possess any more than one instance of the same character, at
    position within the string which is odd.
- Do not possess any more than one instance of the same character, at
    position within the string which is even.

will the above always be the case?
```

```
Example: If given "akbjciakbjdiakbjei", is i encrypted to c, d or e?
```

A: to i, as it does not appear at any even position. Since this encoding is not injective, we'd only expect it to work for encrypt, not decrypt.

Q40

Q: In part 3, the Caesar Cipher, are we to account for a negative integer input as we were required to in Assignment 1?

A: Yes

Q41

Q: “ On assignment 2 part 2, it says that the Substitution class needs 2 constructors. But I’m not sure how to create these constructors. Looking at the UML diagram, the Substitution class has no attributes, so what exactly am I constructing? As far as I’m aware, constructors can’t return values, so I don’t understand how these constructors result in any sort of substitution. ”

A:

- You mean MonoAlphaSubstitution
- You do want some sort of attribute here
- The constructor should initialise this according to its argument

Q42

Q: Can you come to the lab to explain A2?

A: No promises. Talk to me after the lecture or post on canvas if you have questions

Q: What if I have questions about my code?

A: It is an assignment; we cannot help you debug!

Summary

We looked at...

- Abstract classes
- Interfaces
- Polymorphism

Next Week

- Collections
- Iterators
- Generics
- Assignment 2