1 Inheritance

Hierarchies of classes can be created. This is desirable for the following reasons:

- Reusability of code (reusing the functionality of existing classes).
- Code is easier to maintain.
- Polymorphic behaviour (objects are manipulated via reference variables of the base class).

![Inheritance Tree](image)

Figure 1: A inheritance tree of classes in UML form.
Figure 1 shows a hierarchy of classes in UML representation (UML is a modelling language for object oriented programming). A Student is a Person and a PostgraduateStudent is a Student. Similarly, Employee is a Person and Manage is an Employee.

Notice that the is-a attribute is characteristic for the relationship of a class with its parent class.

Student is a subclass (or child class) of Person, and Person is a superclass (or parent class) of Student.

Note that unlike C++, Java does not support multiple inheritance, i.e. a class can only inherit from a single other class.

1.1 Example:

class Person {
    private String name;

    public Person() {
    }

    public Person(String name1) {
        name = name1;
    }

    // initialise the name instance field of the object
    public void setName(String name) {
        /* this is a shortcut for the object we are currently in.
           Thus, this.name is the instance field name within the
           current object */
        this.name = name;
    }

    // prints all information about the object
    public void info() {
        System.out.println("\nname: " + name);
    }
}

class Student extends Person {
    private String school;

    public Student(String school, String name) {
        this.school = school;
        setName(name);
    }

    // prints all information about the object
public void info() {
    // call info() method of Person class
    super.info();
    System.out.println("school: " + school);
}
}

class PostGraduateStudent extends Student {
    private String firstDegree; // what the first degree was on

    public PostGraduateStudent(String school, String name, String degree) {
        super(school, name); // call constructor of parent class
        firstDegree = degree;
    }

    public void info() {
        super.info(); // call info() method of the parent class
        System.out.println("firstDegree: " + firstDegree);
    }
}

public class University {
    public static void main(String[] args) {
        Student s1 = new Student("IC", "John");
        Student s2 = new Student("MIT", "Helen");
        PostGraduateStudent s3 = new PostGraduateStudent("Westminster", "George", "music");

        s1.info();
        s2.info();
        s3.info();
    }
}

When the above program is run, it displays:

name: John
school: IC

name: Helen
school: MIT

name: George
school: Westminster
firstDegree: music

A subclass inherits from the parent class:

• All its methods.
• All its fields

Part of a subclass object is an object of the parent class, i.e. an subclass object contains a parent class object.

For example, although class Student does not contain a method setName, it is able to call the method, as it inherits it from the parent class Person. This is illustrated in the above code.

• Although a subclass inherits the fields and methods of the superclass, this does not mean that it can access these fields and methods directly. Private fields and methods of a superclass, can only be accessed by the subclass via calling public (or protected) methods of the superclass accessing them.

Field name is private in class Person. The field is inherited by subclass Student which cannot access it directly because of its privateness. However, it can be indirectly accessed by calling the public method setName found in the parent class Person, which assigns the field a value.

More details about what fields and methods of a class can be accessed by another class, will be described in Section 7.

In the University example, the this keyword is used. The this keyword refers to the object we are currently in. Therefore this.name accesses the name instance field of the current object, and this.foo() calls method foo on the current object (foo is synonymous to this.foo()).

The this keyword can also be used to call alternative constructors of the same class from inside a different constructor. Such a call must be the first statement in a constructor. For example in the code below, the call this(y) inside D(String, int), calls the D(String) constructor of the class:

```java
class D {
    String s;
    int i;

    public D(String x) {
        s = x;
    }

    public D(String y, int p) {
        this(y); // calls D(String)
        i = p;
    }
}
```

2 Overriding Methods

When a class defines a method with the same signature as a method in a parent class, the method is overridden in the subclass\(^1\).

In the example of the previous section:

\(^1\)The signature of a method consists of its name and the number, type, order of its arguments.
• **Student** overrides the **info** method with its own implementation (initially it inherits the **Person** version of **info**).

• **PostGraduateStudent** overrides the **info** method with its own implementation (initially it inherits the **Student** version of **info**).

Contrast **overriding** with **overloading** described in a set of previous lecture notes.

### 3 The super keyword

The **super** keyword can be used for three purposes:

- To call a constructor of the parent class. In such a case, the **super** call must be the first statement in the constructor of the subclass:

```java
public PostGraduateStudent(String school, String name, String degree) {
    super(school, name);
    firstDegree = degree;
}
```

The first line in the above constructor, calls the constructor `Student(String, String)` of the parent class.

- To call any method of the parent class, even if that method is overridden in the current class. There is no need to have such a call as the first statement in a method.

- To access a field of the parent class. For example, **super.x** in class **B** accesses the **A** instance field **x**:

```java
class A {
    public int x;
}

class B extends A {
    public int x;

    void foo() {
        int y = super.x; // accesses x within A, NOT x in B!
    }
}
```

Note that **super** keywords cannot be combined together. Thus, **super.super.x** in an attempt to access field **x** of the grandparent class will produce a compiler error.
4 Interfaces

An interface is a “abstract” class which does not have method implementations, and just defines a set of method signatures which have to be implemented by a subclass. Therefore, no objects can be created out of an interface.

An interface is used:

- to define the set of public methods (operations) that an implemented class provides.

A class which “implements” the methods outlined in an interface has to use the `implements` keyword.

Interfaces have the following characteristics:

1. They do not provide an implementation for any of their methods.
2. All of their data and methods are public.
3. All of the fields are static and final.

Example:

According to the following interface `Game`, every concrete game (concrete class derived from `Game`) should have methods `computeScore()` and `printInstructions()` implemented.

```java
import java.util.*;

interface Game {
    int computeScore();
    void printInstructions(); // how to play the game
}

class Blackjack implements Game {
    public int computeScore() {
        System.out.println("Blackjack computeScore() called");

        // calculate and return a random score
        Random randomGenerator = new Random();
        int score = randomGenerator.nextInt(22);

        return score;
    }

    public void printInstructions() {
        System.out.println("Blackjack printInstructions() called");
    }

    public int getNumberOfBalls() {
```
class ComputerGame implements Game {
    String platform;

    public ComputerGame(String platform) {
        this.platform = platform;
    }

    public int computeScore() {
        System.out.println("ComputerGame computeScore() called");
        return 100; // buggy game, always returns 100
    }

    public void printInstructions() {
        System.out.println("ComputerGame printInstructions() called");
    }

    public String getPlatform() {
        return platform;
    }
}

public class GameTest {
    public static void main(String[] args) {
        // Game g = new Game(); // Error! Cannot create an object out of an interface

        ComputerGame g1 = new ComputerGame("Linux");
        System.out.println("Score: "+g1.computeScore());
        g1.printInstructions();

        BlackJack g2 = new BlackJack();
        System.out.println("Score: "+g2.computeScore());
        g2.printInstructions();
    }
}

The above program displays:

ComputerGame computeScore() called
Score: 100
ComputerGame printInstructions() called
BlackJack computeScore() called
Score: 11
BlackJack printInstructions() called

Note that a class can implement more than one interfaces. For example, it is possible to have:

```java
interface Vehicle { /* details omitted */ }
interface Sellable { /* details omitted */ }
interface Repairable { /* details omitted */ }

class Car implements Vehicle, Repairable, Sellable {
    // ... implementation omitted
}
```

5 Abstract Classes

An abstract class is a class which cannot be instantiated (i.e. objects of it cannot be created), and has one or more methods without implementation (abstract methods).

As with interfaces, choosing a class to be abstract or not is a design issue.

Example:

```java
abstract class Instrument {
    String manufacturer;

    public abstract void play();

    public void setManufacturer(String m) {
        manufacturer = m;
    }

    public String getManufacturer() {
        return manufacturer;
    }
}

class Violin extends Instrument {
    public void play() {
        System.out.println("Violin’s melody");
    }
}

class Guitar extends Instrument {
    public void play() {
        System.out.println("Guitar music");
    }
}
public class InstrumentTest {
    public static void main(String[] args) {
        // Instrument i1 = new Instrument(); // Error! Cannot instantiate Instrument
        Violin i2 = new Violin();
        Guitar i3 = new Guitar();

        i2.play();
        i3.play();
    }
}

When the above program is run, it displays:

Violin’s melody
Guitar music

6 Polymorphism

A reference variable of type $X$ (which can be a concrete base class, an interface or an abstract class) can hold an object of any subclass of $X$.

- Because of late binding, the specific class of the object held by the reference variable will be determined at run time, and calls to the appropriate methods will be produced.

Thus assuming that $a$ is a reference variable of a base class, an abstract class or an interface the call:

    a.foo()

will invoke a different $foo()$ method if the object that $a$ points to is an object of the base class or an object of a subclass.

This behaviour is called polymorphism (capability of assuming different forms). Calling a method with the same name and signature on a reference variable, corresponds (at run time) to the invocation of the appropriate method according to the actual class of the object that the reference variable points to.

Example:

The following code uses the classes and interfaces of previous examples:
public class PolymorphismTester {
    public static void main(String[] args) {
        Person p = new Student("wmin", "tom");
        p.info(); // calls info() of class Student

        p = new PostGraduateStudent("stanford", "peter", "cs");
        p.info(); // calls info() of class PostGraduateStudent

        System.out.println();

        Instrument instr = new Guitar();
        instr.play(); // calls play of class Guitar
        instr = new Violin();
        instr.play(); // calls play of class Violin

        System.out.println();

        Game game = new BlackJack();
        // game.getNumberOfBalls(); // Error! getNumberOfBalls not in Game
        game.printInstructions(); // call printInstructions() of BlackJack
        game = new ComputerGame("unix");
        game.printInstructions(); // call printInstructions() of ComputerGame
    }
}

The output of the above program is:

name: tom
school: wmin

name: peter
school: stanford
firstDegree: cs

Guitar music
Violin’s melody

BlackJack printInstructions() called
ComputerGame printInstructions() called

7 Access Specifiers

The details (methods, fields) of the implementation of a class can be hidden from other classes.

The access specifiers public, protected, private can be used to determine the access of classes to the individual methods and fields of another class.

The members of a class (i.e. data and methods) can be:
• **Public**: Everybody can access them.

• **Private**: Only the class itself can access them.

• **Protected**: Only the class itself, its subclasses, and the classes in the same package can access them.

The capability of classes to hide information from other classes is one of the fundamental characteristics of object oriented programming, called *encapsulation*.

If a class member (data or method) does not have an access specifier, then by default it has *package access*, i.e. only the class itself and other classes in the same package can access it.

Besides members of classes, the **public** access specifier can precede the definition of a class. Similarly with class members, a **public** class can be accessed by any other class, whether they are in the same package or not. If no **public** precedes the definition of the class, then the class has package access, and it can only be accessed by classes in the same package. The **protected** and **private** specifiers cannot be used in front of a class definition.

**Example:**

The following classes are defined in two separate files `Book.java` and `AccessSpecifiersExample.java`:

File `Book.java` contains:

```java
package p1;

public class Book {
    public int numberOfPages;
    protected boolean paperback;
    private String colour;
    String subject;
}
```

File `AccessSpecifiersExample.java` contains:

```java
package p1;

class TextBook extends Book {
    void modifyData() {
        numberOfPages = 100;  // OK
        paperback = true;     // OK
        colour = "blue";      // Error! colour is private in superclass
        subject = "computer science";  // OK
    }
}
```

/* class AccessSpecifierExample is in the same package p1 with class Book */

```java
public class AccessSpecifiersExample {
    public static void main(String[] args) {
```
Book b = new Book();
b.numberOfPages = 200; // OK
b.paperback = false; // OK
b.colour = "yellow"; // Error! colour is private in Book
b.subject = "engineering"; // OK
}
}

In the code above, the three classes Book, TextBook, AccessSpecifierExample are defined in the same package p1.

Class TextBook is a subclass of Book. Therefore, inside method modifyData, it can access the fields numberOfPages, paperback, subject of the parent class, but not colour which is private in Book.

Class AccessSpecifierExample is in the same package with Book. Therefore inside method main, it can access the fields numberOfPages, paperback, subject of Book, but not colour which is private in Book.

Note what it means that a subclass has access to the protected members of the parent class. The real meaning of this is, that the inherited protected members in an object of the subclass, can be accessed by the methods of that object. It does not mean, that the methods of an object of the subclass can access the protected members of an object of the base class:

package p2;

import p1.Book; // import class A from package p1

public class D extends Book {
    public static void main(String[] args) {
        D d1 = new D();
        d1.paperback = true; // OK

        Book b2 = new Book();
        b2.paperback = true; // No! Error! paperback has protected access in p1.Book
    }
}