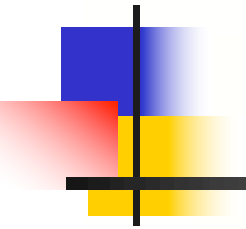


SWEG2031

Object Oriented Programming

Software Engineering Department

AASTU

Chapter 4

Inheritance

Objectives

- Explore the derivation of new classes from existing ones.
- Define and use abstract methods, abstract classes, and interfaces.
- Use super() constructor

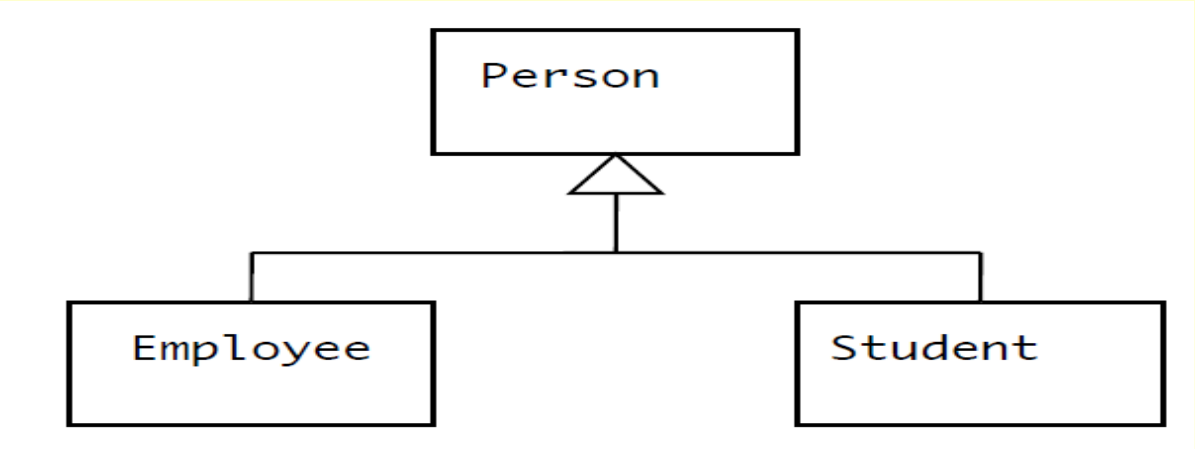
Content

- Inheritance
- Method overloading and overriding
- Abstract classes and Interfaces

Introduction

- Inheritance can be defined as the process where one object acquires the properties of another.
- It allows you to derive new classes from existing classes.
- Allows **reusing** code.
- To avoid redundancy and make the system easy to comprehend and easy to maintain use inheritance.
- Example: Consider people at AASTU. Broadly speaking, they fall into two categories: employees and students.
  - Some features that both employees and students have in common - **name, address, date of birth**, etc.
  - Some features that are unique to each kind of person
    - employee has a pay rate, but a student does not;
    - a student has a GPA, but an employee does not, etc.

Cont'd…



- Employee and Student **inherit all the features** of the class Person **except for the private properties**.
  - In addition, each of the classes Employee and Student can have features of its own not shared with the other classes.
- **Superclass(base class)**: Parent class being extended.
- **Subclass**: Child class that inherits behavior from superclass.
  - A subclass is **not a subset** of its super-class
- **is-a relationship**: Each object of the subclass also "is a(n)" object of the superclass and can be treated as one.

Cont'd…

- "is a" versus "has a"
  - Is-a
    - expresses a relationship between a more special and a more general class
    - A `Student` *is a* `Person`
  - Another relationship is the "has-a"
    - relationship between classes when a class is an attribute of another class.
    - E.g. 1 A car <has-a> engine.
    - E.g. 2 A lecture <has-a> student.

```
public class Car {
    private Engine anEngine;
    private Lights carLights;
    public start () {
        anEngine.ignite ();
        carLights.turnOn ();
    }
}
```

```
public class Engine {
    public boolean ignite () {
      ..  }
}

public class Lights {
    private boolean isOn;
    public void turnOn () {
        isOn = true;}
}
```

Cont'd…

- To inherit a class, you simply incorporate the definition of one class into another by using the **extends** keyword**.**
- Syntax:
  - public class **name** **extends** superclass {
- Example:

```
class Person {
...
}
class Employee extends Person {
...
}
class Student extends Person {
...
}
```

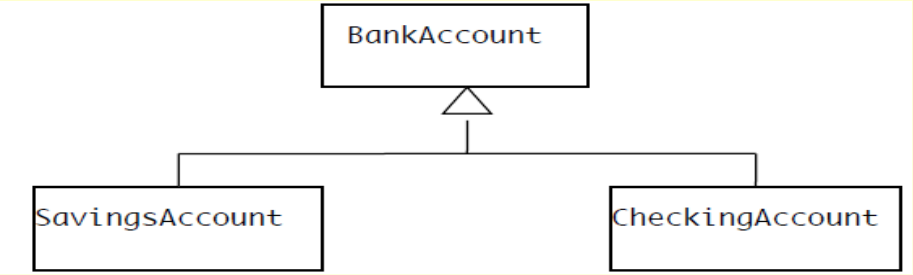
Cont'd…

```
public class Animal{
}
public class Mammal extends Animal{
}
public class Reptile extends Animal{
}
public class Dog extends Mammal{

}
```

- based on the above example, in object oriented terms the following are true:
  - Animal is the superclass of Mammal class.
  - Animal is the superclass of Reptile class.
  - Mammal and Reptile are subclasses of Animal class.
  - Dog is the subclass of both Mammal and Animal classes
- if we consider the IS-A relationship, we can say:
  - Mammal IS-A Animal
  - Reptile IS-A Animal
  - Dog IS-A Mammal
  - Hence : Dog IS-A Animal as well

Example: a class hierarchy for bank accounts



- The classes SavingsAccount and CheckingAccount inherit the features of BankAccount
  - a) Since a BankAccount has an owner and a balance, so does a SavingsAccount or a CheckingAccount.
  - b) Since a BankAccount has methods deposit(), reportBalance(), and getAccountNumber(), so does a SavingsAccount or a CheckingAccount.
- Savings account adds features that an ordinary BankAccount does not have - e.g. payInterest() and setInterestRate().
- CheckingAccount overrides the withdraw() method of BankAccount.
  - a) In the special case where the checking account balance is insufficient for the withdrawal, but the customer has a savings account with enough money in it, the withdrawal is made from savings instead.
  - b) In all other cases, the inherited behavior is used by invoking super.withdraw(amount).

Cont'd…

- In designing a class hierarchy, methods should be placed at the **appropriate** level. Example:
  - deposit(), reportBalance(), and getAccountNumber() are defined in the base class BankAccount, and so are inherited by the two subclasses.
  - If they were defined in the subclasses, we would have to **repeat the code twice** - extra work and an invitation to **inconsistency**.
  - On the other hand, payInterest() and setInterestRate() are defined in SavingsAccount, because they are not relevant for CheckingAccounts.
  - withdraw() is defined in BankAccount and overridden in CheckingAccount. Why is this better than simply defining separate versions in CheckingAccount and SavingsAcccount?

Example

```
// Create a superclass
class A {
   int i, j;
   void showij() {
      System.out.println("i and j: " + i + " " + j);
   }
}
// Create a subclass by extending class A.
class B extends A {
   int k;
   void showk() {
      System.out.println("k: " + k);
   }
   void sum() {
      System.out.println("i+j+k: " + (i+j+k));
   }
}
```

Example(cont'd)

```
class SimpleInheritance {
   public static void main(String args[]) {
     A superOb = new A();
      B subOb = new B();
      // The superclass may be used by itself.
      superOb.i = 10;
      superOb.j = 20;
      System.out.println("Contents of superOb: ");
      superOb.showij();
      System.out.println();
      //The subclass has access to all public members of its superclass
       subOb.i = 7;
       subOb.j = 8;
       subOb.k = 9;
       System.out.println("Contents of subOb: ");
       subOb.showij();
       subOb.showk();
       System.out.println();
       System.out.println("Sum of i, j and k in subOb:");
       subOb.sum();
   }
 }
```

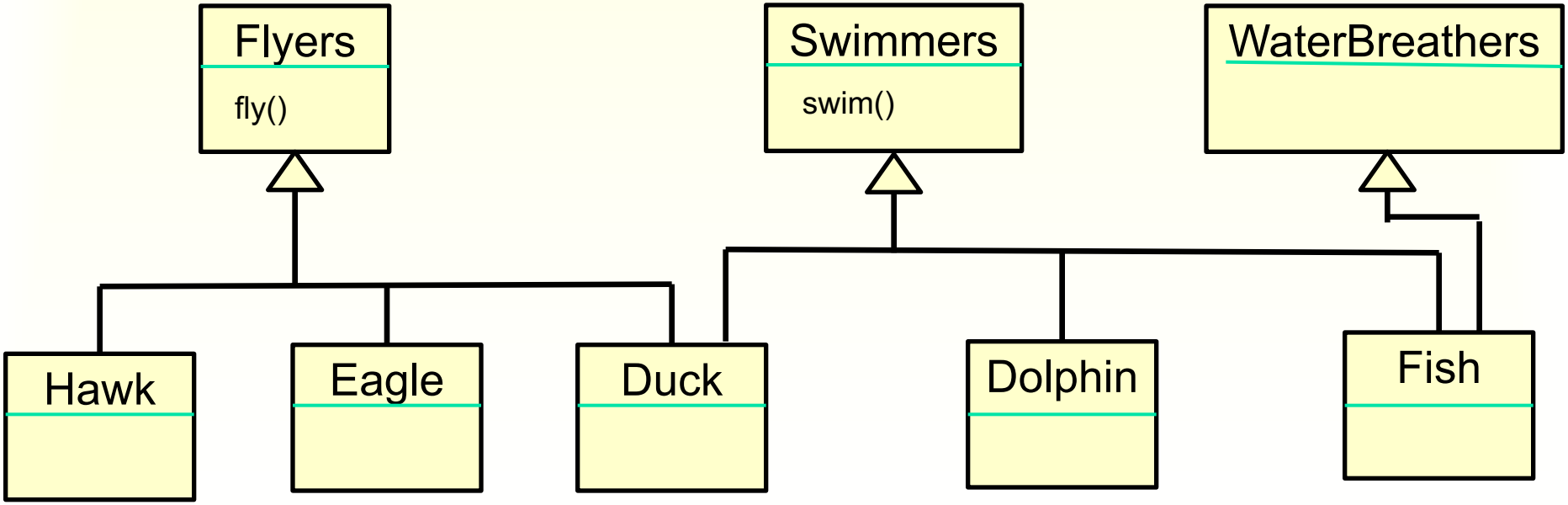
Example(cont'd)

Output:

```
Contents of superOb:
i and j: 10 20
Contents of subOb:
i and j: 7 8
k: 9
Sum of i, j and k in subOb:
i+j+k: 24
```

Multiple Inheritance

- Java's approach to inheritance is called **single inheritance**
- But what happens if some behaviors or attributes are common to a group of classes but some of those classes include behaviors shared with other groups?
- Or some groups of classes share some behaviors but not others?
- Java interfaces provide the best features of multiple inheritance



Activity

- Draw a class diagram showing an inheritance hierarchy containing classes that represent different types of cars

Method Overloading

- Overloading means to define multiple methods with the same name but different signatures.
- Method calls cannot be distinguished by return type if they have the same number and type of parameters.

  E.g. If we define method: **public int square(int no)**

  It is not allowed to define another method like this:

  public double square(int no)

Example 1:

```
public class Foo{
      public void display( ) { }
      public void display(int i) { }
      public void display(char ch) { }
}
Foo f = new Foo ();
f.display( );
f.display(10);
f.display('c');
```

Example 2

```
public class MethodOverload{
   // test overloaded square methods
   public void testOverloadedMethods(){
      System.out.printf( "Square of integer 7 is " +square( 7 ) );
      System.out.printf( "Square of double 7.5 is" + square( 7.5 ) );
   }
   public int square( int intValue ){
    System.out.printf("Called square with int argument:" +intValue );
      return intValue * intValue;
   }
   public double square( double doubleValue ){
      System.out.printf("\nCalled square with double argument:"
  +doubleValue );
      return doubleValue * doubleValue;
   }
}
```

Example 2(cont'd)

```
public class MethodOverloadTest{
   public static void main( String  args[] ){
      MethodOverload methodOverload = new MethodOverload();
      methodOverload.testOverloadedMethods();
   }
}
```

Method Overriding

- Overriding means to provide a **new** implementation for a method in the **subclass**.
- To override a method, the method must be defined in the subclass using the **same signature** and the **same return type**
- Benefit of overriding is ability to define a behavior that's specific to the subclass type which means a subclass can implement a parent class method **based on its requirement**.

Example1

```
class Animal{
    public void move(){
  System.out.println("Animals can move");
    }
}

class Dog extends Animal{
   public void move(){
     System.out.println("Dogs can walk and run");
   }
}
```

Example1(cont'd)

```
public class TestDog{
    public static void main(String args[]){
  Animal a =new Animal();
  Dog b =new Dog();
  a.move(); // runs the method in Animal class
  b.move(); //Runs the method in Dog class
   }
}
```

Program Output

Animals can move

Dogs can walk and run

Example 2

```
class Animal2{
  public void move(){
       System.out.println("Animals can move");
   }
}
class Dog2 extends Animal2{
   public void move(){
      System.out.println("Dogs can walk and run");
   }
   public void bark(){
      System.out.println("Dogs can bark");
   }
}
```

Example2(cont'd)

```
public class TestDog2{
  public static void main(String args[]){
    Animal2 a =new Animal2(); //Animal reference and object
    Animal2 b =new Dog2(); //Animal reference but Dog object
    a.move(); // runs the method in Animal class
    b.move(); //Runs the method in Dog class
    b.bark();
  }
}
```

- **Program output:**

cannot find symbol
symbol : method bark()
location: class Animal
b.bark();

This program will throw a compile time error since b's reference type Animal doesn't have a method by the name of bark

Rules for method overriding

- The **argument list should be exactly the same** as that of the overridden method.
- The **return type should be the same**
- The **access level cannot be more restrictive than the overridden method's access level**. For example, if the superclass method is declared public, then the overriding method in the subclass cannot be either private or protected.
- Instance methods can be overridden only if they are inherited by the subclass.
- A method declared **static cannot be overridden**.
- A subclass within the same package as the instance's superclass can override any superclass method that is not declared private
- A subclass in a different package can only override methods declared public or protected.

The super keyword

- It can be used in two ways:
- **(I) To call a superclass constructor.**
- Each subclass constructor must implicitly or explicitly call its superclass constructor
- Syntax: *super(), or super(parameters);*
- statement super() or super(arguments) must appear in the first line of the subclass.

```
public Student(String initialName, int initialStudentNumber)
{
    super(initialName);
    studentNumber = initialStudentNumber;
}
```

The super keyword(cont'd)

- If a class is designed to be extended, it is better to provide a no-arg constructor to avoid programming errors.
- **Example**:

```
class  Fruit{
 public Fruit(String name) {
     System.out.println("Fruit's constructor is invoked");
   }
 }
public class Apple extends Fruit {
 }
```

- Since no constructor is explicitly defined in Apple, Apple's default no-arg constructor is defined implicitly. Since Apple is a subclass of Fruit, Apple's default constructor automatically invokes Fruit's no-arg constructor. However, Fruit does not have a no-arg constructor, because Fruit has an explicit constructor defined. Therefore, the program cannot be compiled.

The super keyword(cont'd)

- **(II) To call a superclass method**, when invoking a superclass version of an overridden method.
- Syntax: super.method(parameters);

The super keyword(cont'd)

- **Example**

```
class Animal3{
   public void move(){
     System.out.println("Animals can move");
   }
}
class Dog3 extends Animal3{
 public void move(){
   super.move();// invokes the super class method
   System.out.println("Dogs can walk and run");
 }
}
public class TestDog3{
  public static void main(String args[]){
   Dog3 b =new Dog3();
   b.move();//Runs the method in Dog class
  }
}
```

- **program Output**

Animals can move
Dogs can walk and run

Activity 1

What will be an output of the following program?

```
public class Test{
 public static void main(String[] args) {
    A a = new A(3);
  }
 }
 class A extends B {
 public A(int t) {
    System.out.println("A's constructor is invoked");
  }
 }
 class B {
 public B() {
    System.out.println("B's constructor is invoked");
  }
 }
```

Output: B's constructor is invoked
A's constructor is invoked

Activity 2

Identify error in the following program

```
class A {
   public A(int x) {
      System.out.println("A's Constructor");
   }
}
class B extends A {
    public B() {
      System.out.println("B's Constructor");
    }
}
public class C {
    public static void main(String[] args) {
        B b = new B();
     }
}
```

Error:

Can't find constructor A()

Abstract Class

- Class that cannot be instantiated.
- An abstract class must include the keyword abstract in its definition.
- An abstract class has an incomplete definition because the class includes the abstract method that does not have a method body.
- **Format:**

```
public abstract class <class name>{
    <public/private/protected> abstract method ();
}
```

An abstract method

- An ***abstract method is a method with the keyword abstract, and it ends with a semicolon*** instead of a method body.
- If you want a class to **contain a particular method** but you want the **actual implementation** of that method to be determined by child classes, you can declare the method in the parent class as abstract.
- An abstract method **consists of a method signature**, but **no method body**.
- A class is abstract if the class **contains an abstract method** or **does not provide an implementation of an inherited abstract method**.

Example

```
abstract class A {
  abstract void callme();
   //concrete methods are still allowed in abstract classes
   void callmetoo() {
    System.out.println("This is a concrete method.");
  }
}

class B extends A {
  void callme() {
    System.out.println("B's implementation of callme.");
  }
}
```

Example(cont'd)

```
class AbstractDemo {
   public static void main(String args[]) {
        B b = new B();
        b.callme();
       b.callmetoo();
   }
}
```

Output

B's implementation of callme

This is concrete method

Interfaces

- Using interface, you can specify **what a class must do**, but **not how it does it**.
- Once an interface has been defined, one or more classes can implement that interface.
- An interface **is not a class**. A class describes the attributes and behaviors of an object. An interface **contains behaviors that a class implements**.
- An interface is similar to a class in the following ways:
  - An interface can contain any number of methods.
  - An interface is written in a file with a **.java extension,** with the **name of the interface matching the name of the file**.
  - The bytecode of an interface appears in a **.class file.**

Interfaces(cont'd)

- interface is different from a class in several ways:
  - You **cannot instantiate** an interface.
  - An interface **does not contain any constructors**.
  - **All** of the **methods in an interface are abstract**.
  - An interface **cannot contain instance fields**. The only fields that can appear in an interface must be declared as **static**.
  - An interface is **not extended by a class**; it is **implemented** by a class.

Defining an Interface

- General form of an interface:

```
access interface NameOfInterface
{
    //Any number of final, static fields
    //Any number of abstract method declarations
}
```

  - *access is either **public** or **not used(default)***

- Variables can be declared inside of interface declarations. They are implicitly **final** and **static, meaning they cannot be changed by the implementing class.**

- Example of an interface definition:

```
interface Callback {
   void callback(int param);
}
```

Implementing Interfaces

- To implement an interface, include the **implements clause in a class definition,** and then create the methods defined by the interface.
- The general form:

```
access class classname [extends superclass]
   [implements interface [,interface...]] {
   // class-body
}
```

- If a class implements more than one interface, the interfaces are separated with a comma.

Example

```
interface AnimalInterface{
   public void eat();//abstract method
   public void travel();//abstract method
}
```

```
// class implementing the Animal Interface
public class MammalInt implements AnimalInterface{
  public void eat(){
    System.out.println("Mammal eats");
  }
  public void travel(){
     System.out.println("Mammal travels");
  }
}
```

Example(cont'd

```
public class InterfaceDemo{
  public static void main(String args[]){
    MammalInt m = new MammalInt();
    m.eat();
    m.travel();
  }
}
```

Output:
```
Mammal eats
Mammal travels
```

Inheritance versus Interface

- They are similar because they are both used to model an IS-A relationship.
- We use the Java interface to share common behavior (defined by its abstract methods) among the instances of **unrelated classes**.
- We use inheritance, on the other hand, to share common code (including both data members and methods) among the instances of **related classes**.

Abstract class Vs. Interface

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static** and **final variables**.
4) Abstract class **can have static methods, main method and constructor**.	Interface **can't have static methods, main method or constructor**.
5) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
6) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.

7) **Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

The `Final` Modifier

- the keyword `final` means unchanging (used in conjunction with the declaration of constants)
- Methods preceded by the final modifier cannot be overridden

  e.g., `public` ***`final`*** `void displayTwo ()`
- Classes preceded by the final modifier cannot be extended
  - e.g., ***`final`*** `public class ParentFoo`

Exercise

- Design and implement a set of classes that define the employees of a hospital: doctor, nurse, administrator, surgeon, receptionist, janitor, and so on. Include methods in each class that are named according to the services provided by that person and that print an appropriate message.
- Design and implement a set of classes that keep track of various sports statistics. Have each low-level class represent a specific sport.