ABSTRACT CLASS, INTERFACE

CSI 203: OBJECT ORIENTED PROGRAMMING

Tanjina Helaly

Meaning of Abstract

- Adjective
  - existing in thought or as an idea but not having a physical or concrete existence.
- Verb:
  - consider something theoretically or separately from (something else).

Abstraction

- Abstraction is a process of hiding the implementation details from the user, **only the functionality will be provided** to the user.
- In other words, the user **will have the information on what the object does** instead of **how** it does it.
- In Java, abstraction is achieved using
  - Abstract classes and
  - interfaces.

Abstraction

- In OOP (Object Oriented Programming) , Abstraction facilitates the easy conceptualization of real world objects into the software program.
- Humans manage complexity through abstraction.
  - Think about a car.
    - Do you think of a car as a set of tens of thousands of individual parts?
    - No, we think of it as a well-defined object with its own unique behavior.
- This abstraction allows people to use a car without being overwhelmed by the complexity of the parts that form the car.

Abstract Class

- Abstract classes
  - Are **superclasses** (called abstract superclasses) that
    - Cannot be instantiated
    - Incomplete
      - subclasses fill in "missing pieces“
    - Contains zero or more abstract method.
- Contains zero or more abstract methods
  - Concrete subclasses must override
  - Enforce child class to override that method
- Instance variables, concrete methods of abstract class
  - subject to normal rules of inheritance

Purpose of Abstract Class

- Declare common attributes …
- Declare common behaviors of classes in a class hierarchy
- Via Abstract method, it enforce child class to override that method
- Restrict creating object
    - Classes that are too general to create real objects

Abstract Class

- Used only as abstract superclasses for concrete subclasses and to declare reference variables
- Many inheritance hierarchies have abstract superclasses occupying the top few levels

Keyword - abstract

- Use to declare a class **`abstract`**
- Also use to declare a method **`abstract`**
- All **concrete** subclasses must override all inherited abstract methods
  - If any subclass doesn’t implement a superclass’s abstract methods, the child class should also declared `abstract`

Abstract Class– how to declare

```
public abstract class NameOfClass {
		// Any number of instance variables, constructors, concrete methods
		// Zero(0) or more abstract method declarations
}
```

Note:

- Abstract class has all the features and use of normal/concrete class. Only differences are
  - Can not be instantiated
  - Must create a Child class
  - Can contain abstract method
    - If there are abstract methods, child class must override those methods.

Abstract Class– example (no abstract method)

```
abstract class Animal{
      // instance variables
      String name, color;
      double weight;

      // Constructors
      Animal(){ }

      Animal(String name, String color){
          this(name,color, 0.0);
      }

      Animal(String name, String color, double weight){
            this.name = name;
            this.color = color;
            this.weight = weight;
    }

      // Concrete methods
      public void eat(){
          System.out.println(name + " eats.");
      }
}
```

Abstract Class– example(with abstract method)

```
abstract class Animal{
      // instance variables
      String name, color;
      double weight;

      // Constructors
      Animal(){ }

      Animal(String name, String color){
          this(name,color, 0.0);
      }

      Animal(String name, String color, double weight){
            this.name = name;
            this.color = color;
            this.weight = weight;
    }

      // Concrete methods
      public void eat(){
          System.out.println(name + " eats.");
      }

      // abstract methods
      public abstract void makeSound();
}
```

Extending an abstract class

- A class normally extends an abstract class.
- When a class extends an abstract class, the **class must override all the abstract methods** declared in the interface.
- If a class does not override all the behaviors/methods of the abstract parent class, the class must declare itself as abstract.

Extending an abstract class - example

```
class Bird extends Animal{
    public Bird() {
            name = "Bird";
    }

    @Override
    public void makeSound() {
            System.out.println("Chirp");
    }
}

class Tiger extends Animal{
    public Tiger() {
            name = "Tiger";
    }
    @Override
    public void makeSound() {
            System.out.println("Roar");
    }
}
```

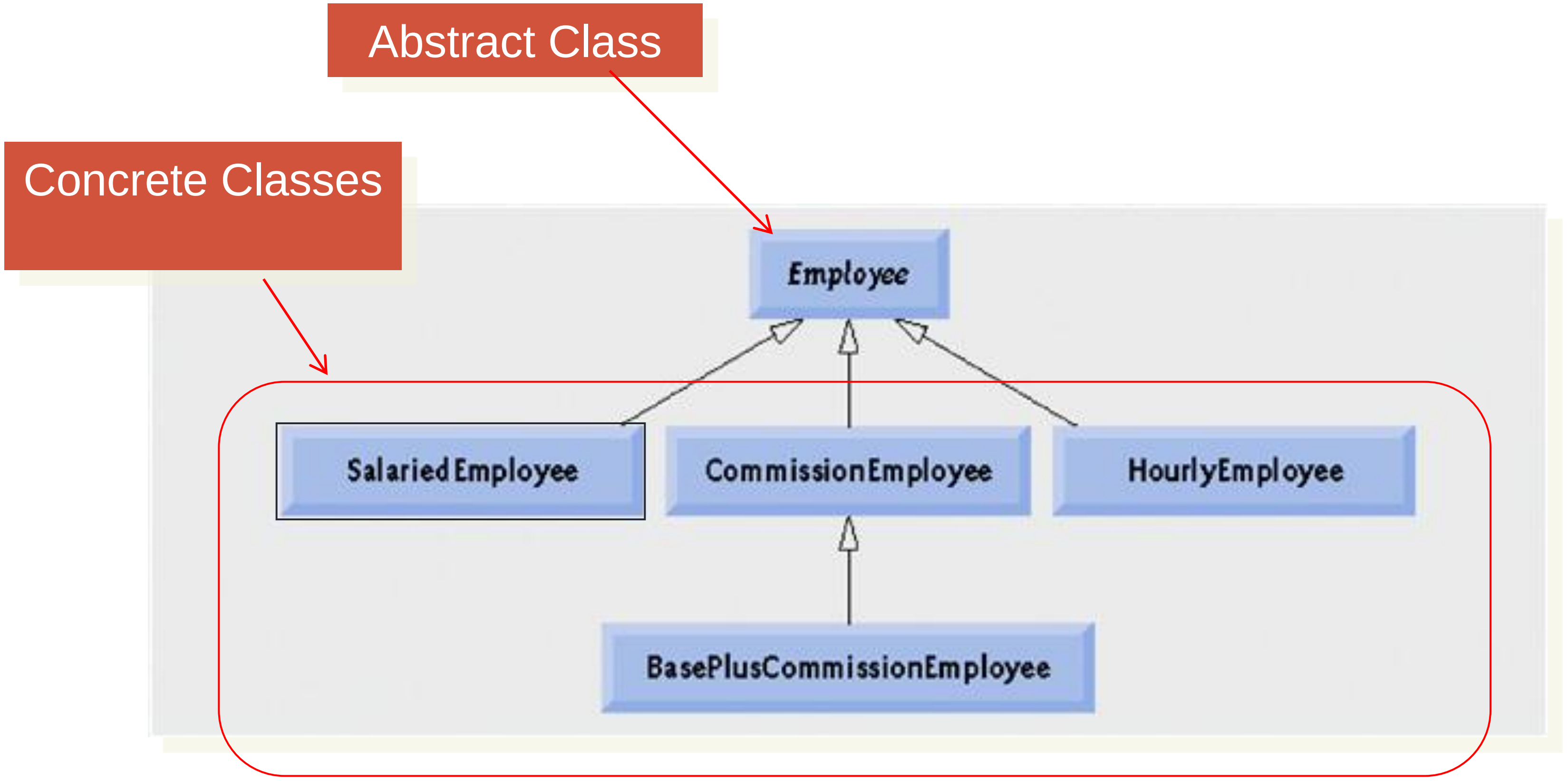
Extending an abstract class - example

```
public class TestAbstractClass {
    public static void main(String[] args) {
        Animal b = new Bird();
        Animal t = new Tiger();
        b.eat();
        t.eat();
        b.makeSound();
        t.makeSound();
    }
}
```

Output:

```
Bird eats.
Tiger eats.
Chirp
Roar
```

Abstract Class – another example



Abstraction vs. encapsulation

- Abstraction represent taking out the behavior from how exactly its implemented,
  - one example of abstraction in Java is interface
- Encapsulation means hiding details of implementation from outside world so that when things change no body gets affected.
  - One example of Encapsulation in Java is private methods; clients don't care about it, You can change, amend or even remove that method if that method is not encapsulated and it were public all your clients would have been affected.

Abstraction vs. encapsulation

- Encapsulation(hiding complexity) implements of abstraction(show what is only necessary)
- Abstraction is the thought process or model
- Encapsulation is the implementation

Interface

Interface

- Using the keyword **interface, you can fully abstract a class' interface from its implementation.**
- That is, using **interface, you can specify what a class must do, but not how it does it.**
- Once it is defined, ***any number of classes can implement an interface.***
- **Also, one class can implement any number of interfaces**.

Interface

- **Interfaces** are syntactically similar to classes, but they can only
    - Have **fields** that are **final** and **static. (**even if they are not explicitly declared as such.**)**
    - Can contains only **public abstract methods. (**even though the interface might not say so)
- Interfaces have the same access levels as classes, public and package.
- An interface, like a class, defines a type.
    - Fields, variables, and parameters can be declared to be of a type defined by an interface.

Interface – What can’t do

- Interfaces can not have
  - instance variables.
    - All fields in an interface are **final** and **static** even if they are not explicitly declared as such.
  - Constructor
  - Normal/concrete method
- Like abstract class, one can not make an object from an interface.

Interface – how to declare

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

Note:

Example

```
interface Flyable{
    public static final String media = "Sky";

    public abstract void fly();
    public abstract boolean needFuel();
}
```

Interface - how to declare

- Each method in an interface is implicitly abstract and public, so the abstract keyword and public access modifier may not explicitly mentioned/declared.

- Each field in an interface is implicitly public, static and final. So those keyword may not explicitly mentioned/declared.

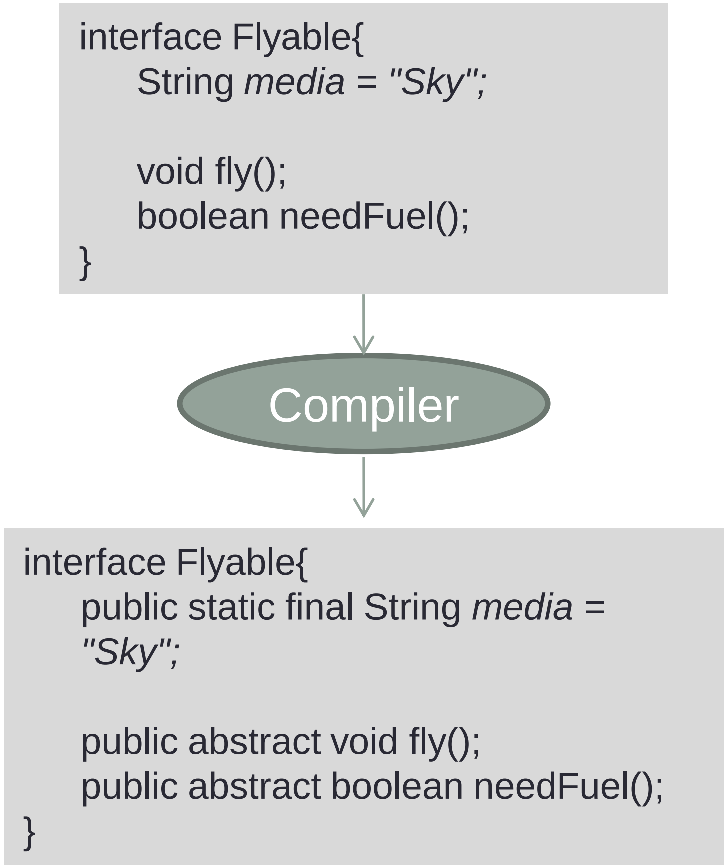
- So, the Flyable interface can be define as below (without the keywords).

```
interface Flyable{
    String media = "Sky";

    void fly();
    boolean needFuel();
}
```

Interface - how to declare

- Compiler do the following conversion



```
interface Flyable{
    String media = "Sky";

    void fly();
    boolean needFuel();
}
```

```
interface Flyable{
    public static final String media =
    "Sky";

    public abstract void fly();
    public abstract boolean needFuel();
}
```

Implementing interface

- A class uses the **implements** keyword to implement an interface.
- When a class implements an interface, you can think of the class as signing a contract, agreeing to perform the specific behaviors/methods of the interface.
  - Which means the **class must override all the methods** declared in the interface.
- If a class does not perform all the behaviors/methods of the interface, the class must declare itself as abstract.

Implementing interface - example

```
class Bird implements Flyable{
    @Override
    public void fly() {
            System.out.println("Bird can fly in the " + Flyable.media);
    }

    @Override
    public boolean needFuel() {
            return false;
    }
}
```

Implementing interface - example

```
class Airplane implements Flyable{
    @Override
    public void fly() {
    System.out.println("Plane can fly in the " + Flyable.media);
    }

    @Override
    public boolean needFuel() {
    return true;
    }
}

public class TestInterface {
    public static void main(String[] args) {
        Bird b = new Bird();
        Airplane a = new Airplane ();
        a.fly();
        b.fly();
    }
}
```

Output:

```
Plane can fly in the Sky
Bird can fly in the Sky
```

Implementing interface – some rules

- A **class** can implement more than one interface at a time.
  - Each interface name is separated by comma after the implements keyword.
  - The class **must override** all methods of all interfaces.
- A **class** can both extends a class and implements many interfaces.
- An **interface** can **extend** other **interfaces** (allow multiple extends), in a similar way as a class can extend another class.
- If a **parent class** implements an interface, its **child classes** automatically implement the interface.

Implementing multiple interfaces - example

```
interface Flyable{
      public static final String media = "Sky";

      public abstract void fly();
      public abstract boolean needFuel();
}

interface Floatable{
    public abstract void canFloat();
}

class Bird implements Flyable, Floatable{
      public void fly() {
          System.out.println("Bird can fly in the " + Flyable.media);
      }

      public boolean needFuel() {
          return false;
      }
      public void canFloat() {
          System.out.println("Bird can float in air.");
      }
}
```

Example – with both extends and implements

```
interface Flyable{
      public static final String media = "Sky“;
      public abstract void fly();
}

interface Floatable{
    public abstract void canFloat();
}


class Animal{
    String name, color;
}

class Bird extends Animal implements Flyable, Floatable{
      public void fly() {
          System.out.println("Bird can fly in the " + Flyable.media);
      }

      public void canFloat() {
          System.out.println("Bird can float in air.");
      }
}
```

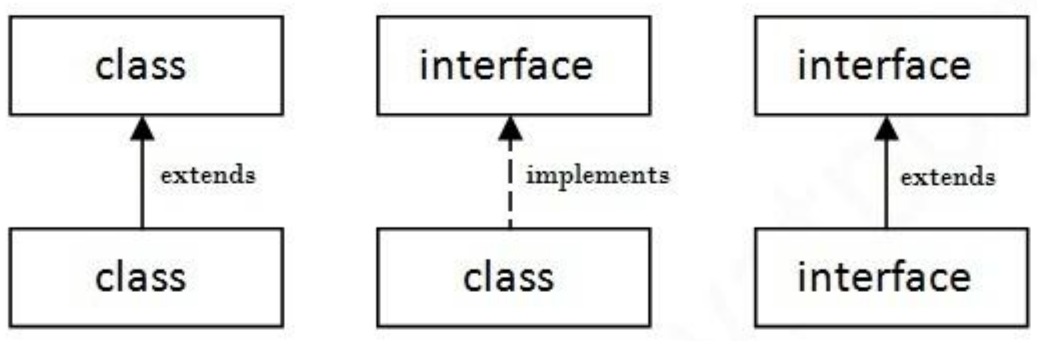
“extends” should appear before “implements”

Extending multiple interfaces

- A Java **class** can only extend one parent class. Multiple inheritance is not allowed.
- **Interfaces** are not classes, however, and an interface can extend more than one parent interface.
- The **extends** keyword is used once, and the parent interfaces are declared in a **comma-separated** list.
- For example,
  - if the Hockey interface extended both Sports and Event, it would be declared as −
  - **public interface Hockey extends Sports, Event**

Relationship between classes & interfaces

- As shown in the figure given below, a class extends another class, an interface extends another interface but a **class implements an interface**.



Why use Interface

- Java has single inheritance, only
  - This means that a child class inherits from only one parent class
  - Sometimes multiple inheritance would be convenient
  - *Interfaces* give Java some of the advantages of multiple inheritance without incurring the disadvantages
- Provide capability for unrelated classes to implement a set of common methods
  - Implementing an interface is a "**promise**" to include the specified method(s)
- Define and standardize ways people and systems can interact.

Interface as data type

- An interface, like a class, defines a **type**.
  - Fields, variables, and parameters can be declared to be of a type defined by an interface.
  - Java Interface also **represents IS-A relationship**. So, interface can be used for subclass polymorphism.
- Remember the 2 classes(Birds and Airplane) that implemented Flyable interface,
  - we can create object of Bird or Airplane and assign those to a Flyable variable as shown below.

```
public class TestInterface {
    public static void main(String[] args) {
        Flyable b = new Bird();
        Flyable a = new Airplane ();
        a.fly();
        b.fly();
    }
}
```

When to use - Abstract Class vs. Interface

- Let B & C be classes. Assume both B and C has some commonalities.
- So, we make A the parent class of B and C.
  - A can hold the methods and fields that are common between B and C.

$$\begin{tikzcd}
 & \text{A (Concrete Class)} \arrow[dl, dash] \arrow[dr, dash] & \\
\text{B} & & \text{C}
\end{tikzcd}$$

When to use - Abstract Class vs. Interface

- If a method in B is so different from the same method in C, there is no shared implementation possible in A.
    - We can make the method and A an abstract classes. The methods in A then indicate which methods must be implemented in B and C. A can act as type, which can hold objects of type B or C.

$$\begin{tikzcd}
 & \text{A (Abstract Class)} \arrow[dl, dash] \arrow[dr, dash] & \\
\text{B} & & \text{C}
\end{tikzcd}$$

When to use - Abstract Class vs. Interface

- But if **all** the methods of B must be implemented differently than the same method in C. And there is **no common attributes** between B and C, make A an **interface.**

$$\begin{tikzcd}
 & \text{A (interface)} \arrow[dl, dash] \arrow[dr, dash] & \\
\text{B} & & \text{C}
\end{tikzcd}$$

Reference

- Java: Complete Reference: Chapter 7,8, 9
- Java: How to Program: Chapter 10

- Online Reference:
  - https://www.tutorialspoint.com/java/java_interfaces.htm
  - http://www.javatpoint.com/interface-in-java
  - https://www.youtube.com/watch?v=1Q4I63-hKcY
  - https://www.youtube.com/watch?v=yyU3bXyc_oU