Object Oriented Programming

ICT2122

Interfaces

P.H.P. Nuwan Laksiri
Department of ICT
Faculty of Technology
University of Ruhuna

Lesson 04

Recap - Object Oriented Concepts

- Object Oriented Programming simplifies the software development and maintenance by providing some concepts,
  - Object
  - Class
  - Inheritance
  - Polymorphism
  - Abstraction
  - Encapsulation

Outline

- Interface - Introduction
- Interface Syntax
- Interface – Hands - On
- Interface - Example
- Interface vs Abstract Class
- Interface – Strict Rules
- Implementing Interfaces
- Adding Fields to an Interface
- Extending Interfaces
- Using an Interface as a Type
- Using Interfaces for Callbacks
- Using default methods
- Using static methods
- Key Things to Remember
- UML - Interface Representation

Interface - Introduction

Interface - Introduction

- 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.**

Interface - Introduction

- An interface is similar to an abstract class, but an interface can include only
  - abstract methods and
  - final fields (constants)
- An interface can't be used as a base class.
- A class implements an interface by providing code for each method declared by the interface.
- A class can extend only one other class, but it can implement as many interfaces as you need.

Interface - Introduction

- An **Interface is a contract**.
- An **Interface is a reference type**, similar to a class, that can **contain only**
  - **Constants**
  - **Method Signatures**
- In addition to abstract methods, interfaces can also define
  - default and static methods in Java 8 and above,
  - as well as nested types and annotations.
  - Method bodies exist only for default methods and static methods.
- Interfaces cannot be instantiated
  - They can only be **implemented by classes** or **extended by other interfaces**.

Interface - Introduction

Interface - Syntax

```
AccessModifier interface name
{
        //public static final variables(Constants)
        type varname1 = value;
                                .
                            .
         type varnameP = value;

         //public abstract methods
        return-type methodname1(para-list);
                            .
                            .
        return-type methodnameN(para-list);
}
```

Interface – Hands - On

```
// Define an interface
public interface Printable {
    void print();
}
```

```
// Implement the interface for a MyDocument
public class MyDocument implements
Printable {
    public void print() {
        System.out.println("Printing the document...");
    }
}
```

```
// Use the implemented printable interface
public class Test {
   public static void main(String[] args) {
            MyDocument doc = new MyDocument();
            doc.print();
    }
}
```

Interface - Declaration

- The following is a legal declaration:

```
public abstract interface Rollable{ };
```

- Both of these declarations are legal and functionally identical:

```
public abstract interface Rollable{};
public interface Rollable{}; //encouraged
```

- Homework
  - What do u you mean by a “tag” or “marker” interface
  - Why we need to use them?

Interface - Example

```
public interface Bounceable
{
        void bounce();
        void setBounceFactor(int bf);
}

public abstract interface Bounceable
{
        public abstract void bounce();
        public abstract void setBounceFactor(int bf);
}
```

Interface vs Abstract Class

- **Interface is a 100% abstract class.**
- Like an abstract class, an interface defines abstract methods that take the following form.

```
abstract void bounce();
```

  - Ends with a semicolon rather than curly braces.
- But while an **abstract class can define both abstract and non-abstract methods**, an **interface can have only abstract methods.**

Interface vs Abstract Class

- Another way interfaces differ from abstract classes is that interfaces have very little flexibility in how the methods and variables defined in the interface are declared.
- All interface methods are implicitly public and abstract.
- In other words, you do not need to actually type the public or abstract modifiers in the method declaration, but the method is still always public and abstract.

Interface – Strict Rules

- Since interface methods are abstract, they cannot be marked final.
- An interface can extend one or more other interfaces.
- An interface cannot extend anything but another interface.
- An interface cannot implement another interface or class.
- An interface must be declared with the keyword interface.

Homework

- What is a “Nested Interface” in JAVA?
- Find some examples for “Nested Interfaces”.
- List down the advantages and disadvantages of “Interfaces” in JAVA.

Implementing Interfaces - Syntax

- To implement an interface, a class must do two things:
  - It must specify an implements clause on its class declaration.
  - It must provide an implementation for every method declared by the interface.

accessmodifier class ***implements*** inter-name
{
    //class body
}

accessmodifier class ***implements*** inter1, inter2,……. InterN
{
    //class body
}

Implementing Interfaces - Example

```
interface bounceable
{
        void bounce();
        void setBounceFactor(int bf);
}

class Tire implements Bounceable
{
        public void bounce(){..}
        public void setBounceFactor(int bf){..}
}
```

Implementing Interfaces - Example

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

public class Client implements Callback{
        public void Callback(int p){
                System.out.println("callback called with" + p);
        }
        void nonIfaceMeth(){
                System.out.println("define its own method");
        }
}
```

Implementing Interfaces - Example

- If an interface is not fully implemented by the implementing class, the class must be declared as abstract.

```
interface Callback
{
        void callback(int param);
}
abstract class CallbackAbs implements Callback
{
        int a, b;
        Void show() {
                System.out.println(a + “” +b);
        }
// …..
}
```

Interface methods can declare with any combination

- The following five method declarations, if declared within their own interfaces, are legal and identical.
  - void bounce();
  - public void bounce();
  - abstract void bounce();
  - abstract public void bounce();
  - public abstract void bounce();

The following interface method declaration won't compile:

- final void bounce(); X
  - //final and abstract can never be used together, and abstract is implied
- private void bounce(); X
  - //interface methods are always public
- protected void bounce(); X
  - //interface methods are always public
- static void bounce(); X
  - //interfaces define instance methods in this way,

    default and static methods should contain the implementation

Adding Fields to an Interface

- All **Fields** defined in an interface **must be public, static and final.**
- In other words, interfaces can declare **only constants, not instance variables.**

Identify the problem with the following code

```
interface Foo
{
        int BAR=42;
        void go();
}
class Zap implements Foo
{
        public void go(){
                BAR=27;
        }
}
```

Look for interface definitions that define constants, but without explicitly using the required modifiers

- public int x=1; //Looks non-static and non-final, but isn't
- int x=1; //Looks default, non-static, non-final, but isn't
- static int x=1; //Doesn't show final or public
- final int x=1; //Doesn't show public or static
- public static int x=1; //Doesn't show final
- public final int x=1; //Doesn't show static
- static final int x=1; //Doesn't show public
- public static final int x=1; //what you get implicitly

- Any combination of the required (but implicit) modifiers is legal, as issuing no modifiers at all!

Extending Interfaces (Interface Inheritance)

- An interface can extend interfaces by using the **extends** keyword.
- An interface that extends an existing interface is called a **subinterface**, and the interface being extended is called the **superinterface**.
- When you use the extends keyword with interfaces,
  - **all the fields and methods** of the superinterface are effectively copied into the subinterface.
- The subinterface consists of a combination of the fields and methods in the superinterface plus the fields and methods defined for the subinterface.

Extending Interfaces (Interface Inheritance)

```
public interface ThrowableBall
{
        void throwBall();
        void catchBall();
}
```

```
public interface KickableBall
{
        void kickBall();
        void catchBall();
}
```

```
public interface PlayableBall extends ThrowableBall, KickableBall
{
        void dropBall();
}
```

- When a class implements an interface that extends another interface,
  - the class must provide implementations for all methods in the interface chain.

    OR
  - the class must be declared abstract

Implementing an Inherited Interface - Example

```
public interface A
{
    void methA1();
    void methA2();
}
```

```
public interface B extends A
{
    void methB1();
}
```

```
public class MyClass implements B{
    public void methA1(){
        System.out.println("Implement meth1().");
    }

    public void methA2(){
        System.out.println("Implement meth2().");
    }

    public void methB1(){
        System.out.println("Implement meth3().");
    }
}
```

Using an Interface as a Type

- In Java, an interface is a kind of type, just like we used with class and abstract class.
- As a result, you can use an interface as
  - the type for a variable,
  - parameter, or
  - method return value

Ex : B myClass = new MyClass();

Using Interfaces for Callbacks

- Callback is a mechanism that
  - allows you to pass a function or method as a parameter to another function or method and
  - have the called function invoke the passed-in function at a certain point in its execution.
- This allows you to provide a piece of code that can be executed by another piece of code, without knowing exactly when or how it will be executed.
- The most common use of callbacks is in graphical applications built with Swing,
  - where you create event listeners that handle user-interface events, such as mouse clicks.
- More in GUI development with Swing

```
public interface ActionListener {
        void actionPerformed(ActionEvent e);
}
```

Inherent limitation of Interfaces

- After you define an interface and then build classes that implement the interface, there's no easy way to modify the interface by adding additional methods to it.
- Example ?
- Solution : using Default methods

Using *default* methods

- After Java 1.8, introduces a new type of interface method called a ***default method***,
  - which supplies code that will be used as the implementation of the method for any classes that implement the interface
  - but do not themselves provide an implementation for the default method

```
public interface Playable
{
    void play();
    default void quit()
    {
        System.out.println("Sorry, quitting is not allowed.");
    }
}
```

```
class Game implements Playable
{
        public void play()
        {
            System.out.println("Good luck!");
         }
}
```

Using *static* methods

- You can use the static keyword to define a static method in an interface.
- This is useful for defining utility methods that can be used by all classes that implement the interface.

```
public interface MyInterface {
    void myMethod();
    static void myStaticMethod() {
        System.out.println("This is a static method.");
    }
}
```

Abstract Class vs Interface

	Abstract Class	Interface
1	An abstract class can extend only one class or one abstract class at a time	An interface can extend any number of interfaces at a time
2	An abstract class can extend another concrete (regular) class or abstract class	An interface can only extend another interface
3	An abstract class can have both abstract and concrete methods	An interface can have only abstract methods
4	In abstract class keyword "abstract" is mandatory to declare a method as an abstract	In an interface keyword "abstract" is optional to declare a method as an abstract
5	An abstract class can have protected and public abstract methods	An interface can have only public abstract methods
6	An abstract class can have static, final or static final variable with any access specifier	Interface can only have public static final (constant) variable

Key Things to Remember

(tryout your own examples for below points)

- We can't instantiate an interface in java.
  - That means we cannot create the object of an interface
- Interface provides full abstraction as none of its methods have body.
  - On the other hand, abstract class provides partial abstraction as it can have abstract and concrete(methods with body) methods both.
- "implements" keyword is used by classes to implement an interface.
- While providing implementation in class of any method of an interface, it needs to be mentioned as public.
- Class that implements any interface must implement all the methods of that interface; else the class should be declared abstract.

Key Things to Remember

(tryout your own examples for below points)

- Interface cannot be declared as private, protected or transient.
- All the interface methods are by default abstract and public.
- Variables declared in interface are public, static and final by default.
- Interface variables must be initialized at the time of declaration otherwise compiler will throw an error.

Key Things to Remember

(tryout your own examples for below points)

- Inside any implementation class, you cannot change the variables declared in interface
  - because by default, they are public, static and final.
- An interface can extend any interface but cannot implement it.
  - Class implements interface and interface extends interface.
- A class can implement any number of interfaces.
- If there are two or more same methods in two interfaces and a class implements both interfaces, implementation of the method once is enough.

Key Things to Remember

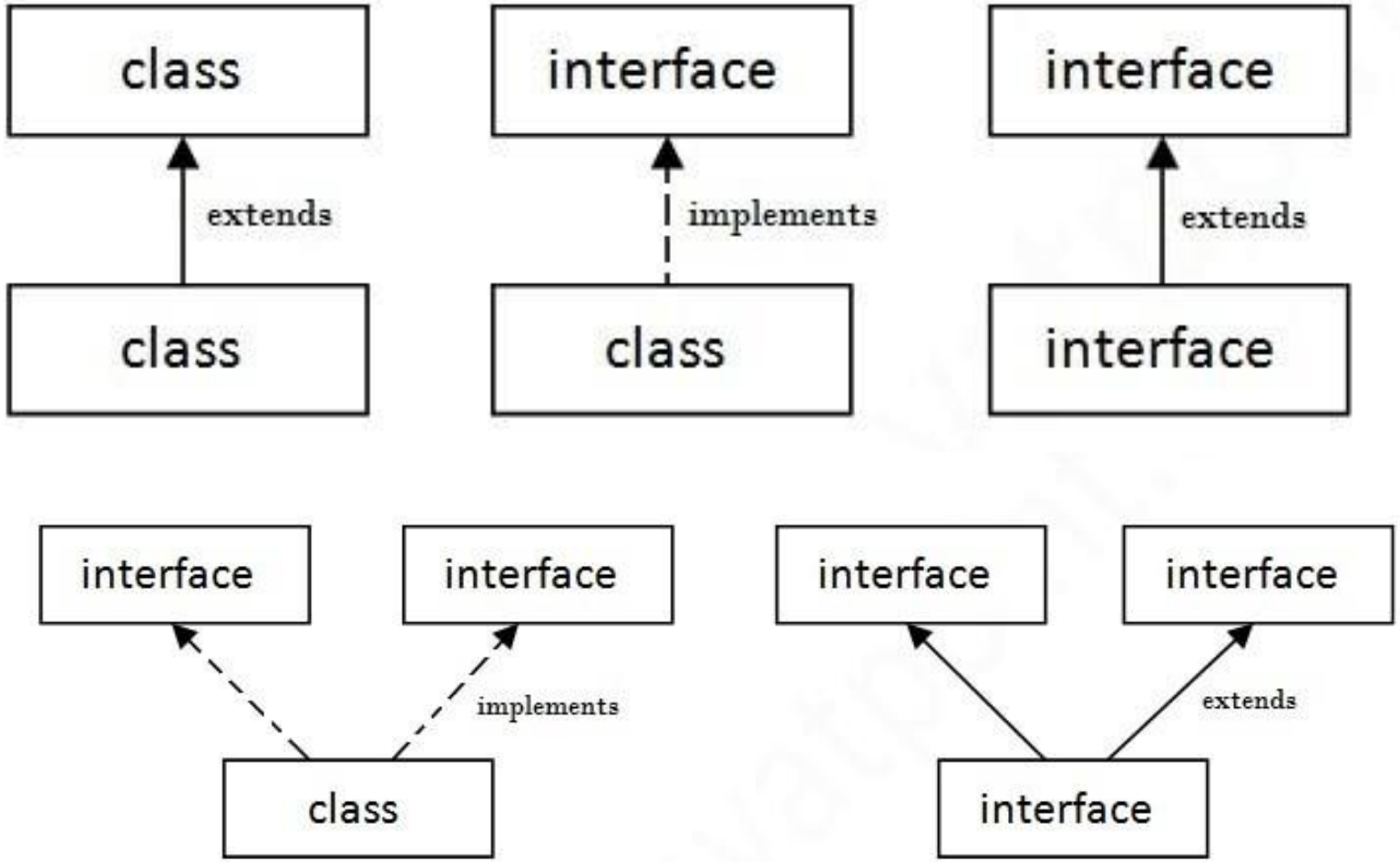
(tryout your own examples for below points)

- A class cannot implement two interfaces that have methods with same name but different return type.
  - Is it ? You must check this.

- Variable names conflicts can be resolved by using interface name.
  - When interface A and B have same variable x, we can access x like below
    - A.x; // A's x
    - B.x //B's x

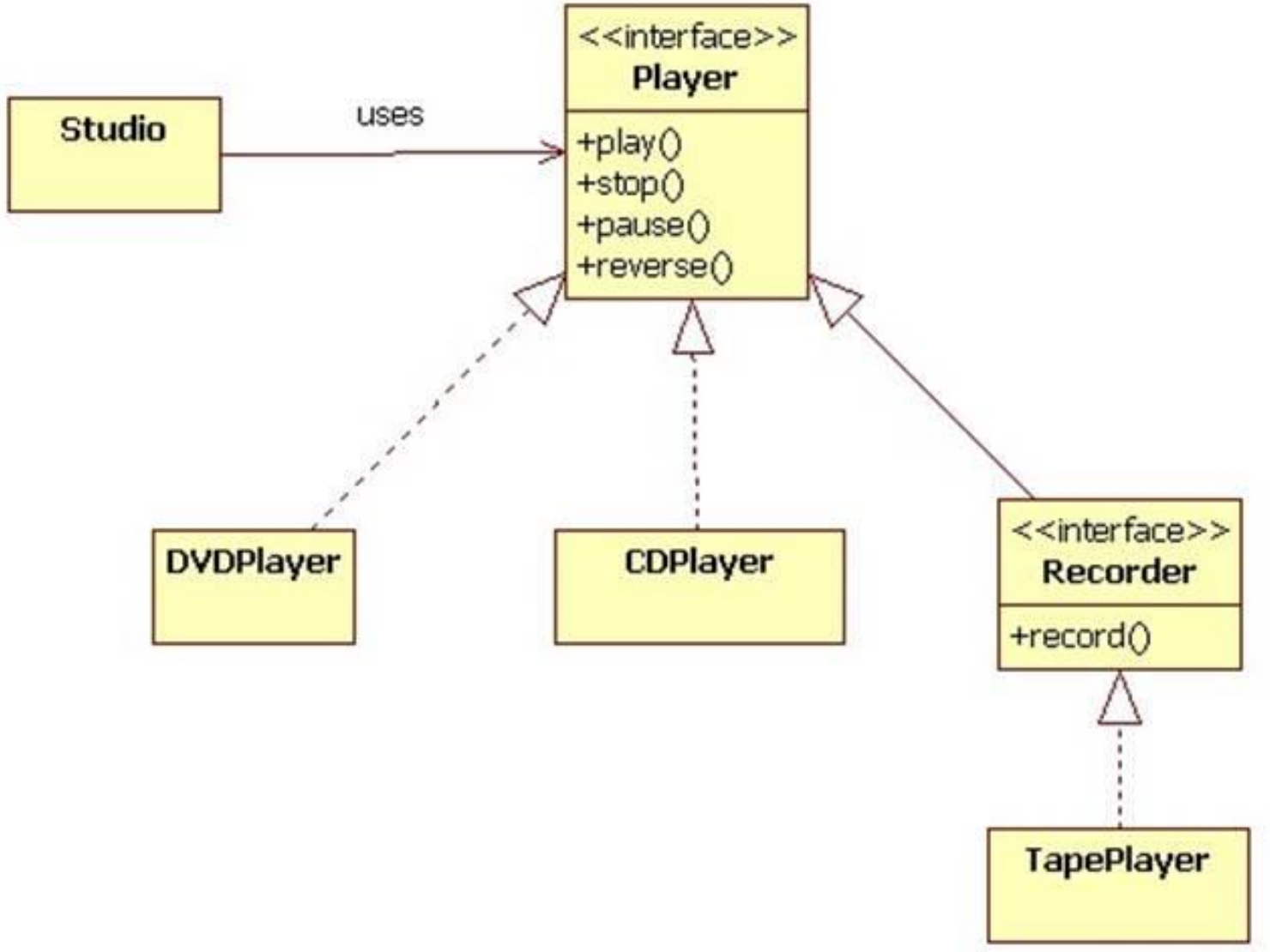
Key Things to Remember

(tryout your own examples for below points)



Multiple Inheritance in Java

UML



http://www.cs.sjsu.edu/~pearce/modules/lectures/oop/basics/interfaces

Summary

- An interface is a collection of abstract methods and constants.
  - It defines a set of methods that a class must implement in order to be considered an implementation of that interface.
- You define an interface using the interface keyword, and you can define any number of methods and constants in the interface.
- Any class that implements an interface must provide an implementation for all the methods defined in that interface.
- You can use an interface as a data type, and you can create objects of classes that implement the interface.
- An interface can extend another interface, which means that it inherits all the methods and constants from the parent interface.
- You can use the default keyword to define a default implementation for a method in an interface.
  - This is useful for providing a default implementation that can be used by all classes that implement the interface.

Summary

- You can use the static keyword to define a static method in an interface.
  - This is useful for defining utility methods that can be used by all classes that implement the interface.
- You can use an interface to define a "marker interface," which is an interface that doesn't have any methods or constants.
  - This is useful for indicating that a class has a particular property, without requiring any methods or constants to be defined.
- You can use an interface to define an event listener interface, which is an interface that specifies methods for handling events.
  - Event listener interfaces are commonly used in graphical user interface (GUI) programming, where the user's actions generate events that need to be handled by the program.
- Interfaces are a powerful tool for defining a contract between different parts of a program, allowing for flexible, modular, and extensible code.
  - By defining an interface, you can separate the definition of a set of methods from their implementation, making it easier to modify and extend your code over time.

References

- https://docs.oracle.com/javase/tutorial/java/landl/createinterface.html

- How To Program (Early Objects)
  - By H .Deitel and P. Deitel
- Headfirst Java
  - By Kathy Sierra and Bert Bates

Questions ???

Thank You