






# Application Program Development

Segment : Interfaces and Implementations

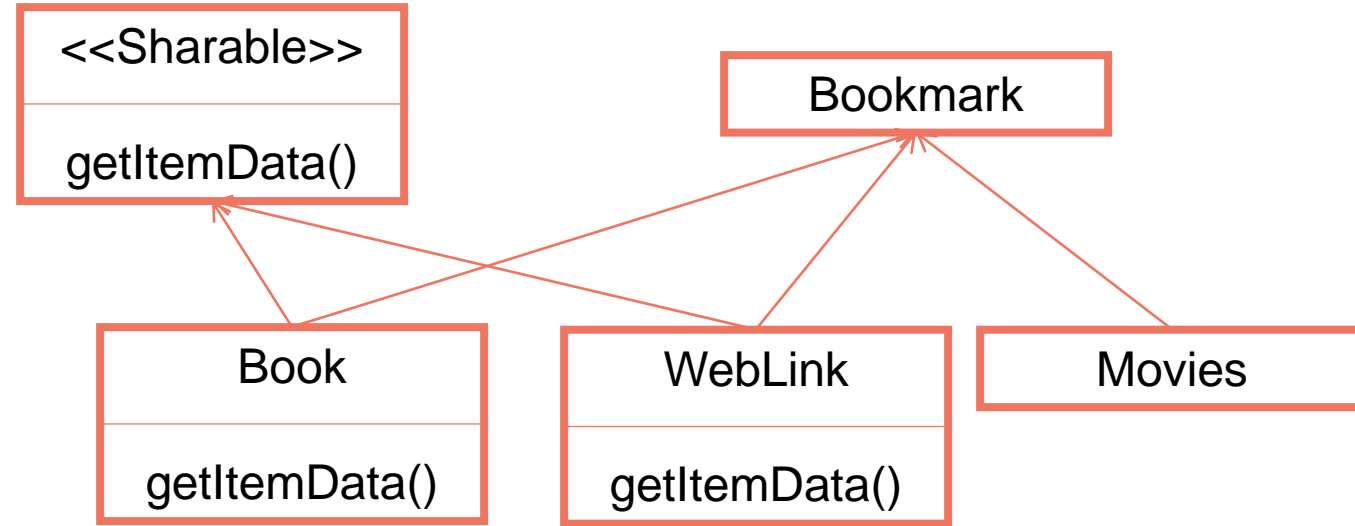


# Outcomes

- Understanding of multiple Inheritance in Java
  - Understanding of Interfaces in Java
- 

- 
- 
- Inheritance allows all classes along the same path in the class hierarchy to share attributes and behaviors.
  - The structure of the class hierarchy helps to identify common behavior that subclasses have with their superclasses.
  - How though, would we define (and perhaps *force*) common behavior between seemingly *unrelated* classes in different parts of the class hierarchy ?

# Multiple Inheritance



- Java does not support multiple inheritance.
- Why?

Diamond Problem

# Interfaces

- An **interface** is a specification (i.e., a list) of a set of methods such that any classes implementing the interface are forced to write these methods.
- An interface is a class like construct that contains only constants and abstract methods.
- In many ways, an interface is similar to an abstract class, but the intent of an interface is to specify common behavior for objects.
- For example, you can specify that the objects are *comparable*, *edible*, *cloneable* using appropriate interfaces.

# Interface Definition

- Interface is a reference type in Java, similar to a class.
- Interfaces cannot be instantiated, they can only be *implemented* by a class or *extended* by other interfaces.

```
interface InterfaceName {  
    constant(s) - final static fields  
    abstract method declaration(s)  
    default method(s)  
    static method(s)  
    nested types  
}
```

An interface creates a new reference data type, just as class definition

```
InterfaceName refVariable;
```

# Interface Structure

```
public interface Sharable{  
    String getItemData();  
}
```

- Abstract method is *public and static* by default (modifiers omitted)
- Fields are *public, static and final* by default (modifiers omitted)
- All members are *public* by default
- Members can't be *private & protected*.

# Abstract Classes VS Interfaces

```
public abstract class Loan {
    public abstract float calculateMonthlyPayment();
    public abstract void makePayment(float amount);
    public abstract void renew(int numMonths);

    public Customer getClientInfo() { // a non-abstract method
        //...
    }
    //....
}
```

## Similarities:

- Three similar methods with no code.
- like abstract classes, we cannot create instances of interfaces. So, we cannot do the following anywhere in our code:

`new Loan()` nor `new Loanable()`

```
public interface Loanable {
    public float calculateMonthlyPayment();
    public void makePayment(float amount);
    public void renew(int numMonths);
}
```

## Differences:

- We cannot declare/define any attributes nor **static** constants in an **interface**, whereas an **abstract** class may have them
- In previous versions of JAVA, we were only able to declare “empty” methods in an **interface**, we could not supply code for them (newer versions of JAVA allow interfaces to have code). In contrast, an **abstract** class in general will often have **non-abstract** methods with complete code.
- All methods in an **interface** must be declared **public**



# Explanation

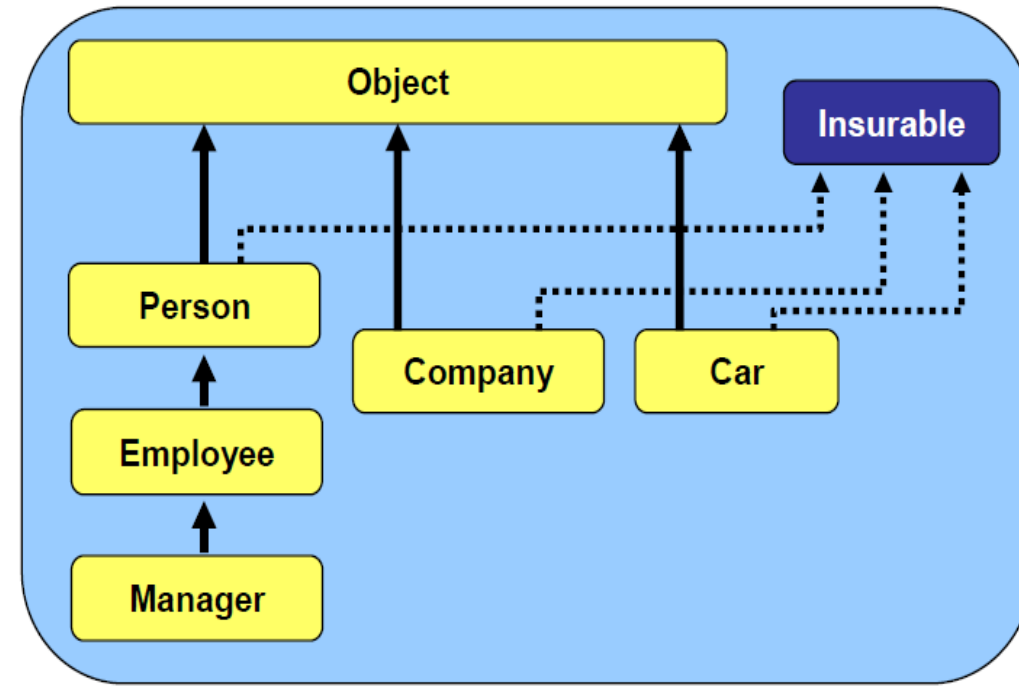
- Assume now that we want to have some classes in our hierarchy that are considered to be insurable.
- Perhaps **Person**, **Car** and **Company** objects in our application are all considered to be **Insurable** objects.
- We simply add the keyword **implements** in the class definition, followed by the **name** of the interface

```
public class Person implements Insurable {  
    ...  
}
```

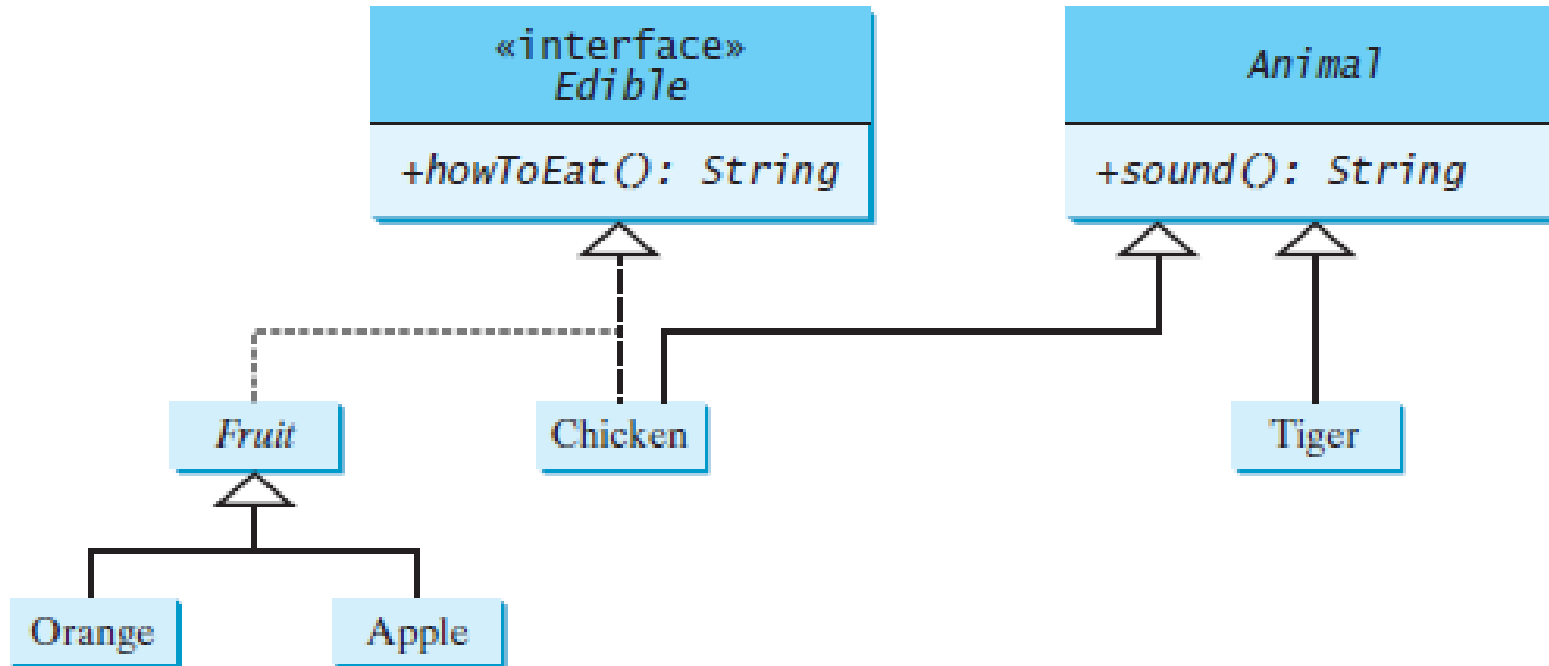
```
public class Company implements Insurable {  
    ...  
}
```

```
public class Car implements Insurable {  
    ...  
}
```

```
public interface Insurable {  
    public int getPolicyNumber();  
    public int getCoverageAmount();  
    public double calculatePremium(int days);  
    public java.util.Date getExpiryDate();  
}
```



# Interfaces Cont....



## <<Edible>>

```
public interface Edible {  
    public abstract String howToEat();  
}
```

## <Animal>

```
public abstract class Animal {  
    public abstract String sound();  
}
```

## Tiger

```
public class Tiger extends Animal{  
    @Override  
    public String sound() {  
        return "Tiger: RROAARR";  
    }  
}
```

## Chicken

```
public class Chicken extends Animal implements Edible{  
  
    @Override  
    public String howToEat() {  
  
        return "Chicken: Fry it";  
    }  
  
    @Override  
    public String sound() {  
  
        return "Chicken: cock-a-doodle-doo";  
    }  
  
}
```

## <Fruit>

```
public abstract class Fruit implements Edible{  
    }  
}
```

## Apple

```
public class Apple extends Fruit{  
  
    @Override  
    public String howToEat() {  
        return "Apple: Make apple cider";  
    }  
}
```

## Orange

```
public class Orange extends Fruit{  
  
    @Override  
    public String howToEat() {  
        return "Orange: Make orange juice";  
    }  
}
```

## TestEdible

```
public class TestEdible {  
  
    public static void main(String[] args) {  
        Object[] objs = {new Tiger(), new Chicken(), new Apple()};  
  
        for(int i=0; i<objs.length; i++) {  
            if(objs[i] instanceof Edible) {  
                System.out.println(((Edible)objs[i]).howToEat());  
            }  
            if(objs[i] instanceof Animal) {  
                System.out.println(((Animal)objs[i]).sound());  
            }  
        }  
    }  
}
```

# Some Common Interfaces of Java API

- **Comparable:**

- Interface Comparable is used to allow objects of the class that implements the interface to be compared to another.
- Common use for ordering the objects in collections like ArrayList.

- **Serializable:**

- Used to identify classes whose objects can be written to (serialized) or read from (deserialized) some type of storage (file, disk, database etc.) or transmitted across a network.

- **Runnable:**

- Used by any class that represents a task to perform, Multithreading.

- **AutoCloseable:**

- Used for closing resources of the current object. Prevents resource leaks.
- Invoked automatically on objects managed by the *try-with-resources* statement.

- **ActionListener:**

- An interface in java.awt package is used to assign behavior to events when a user clicks on a button or other graphical control

# Default Methods in Interfaces

- Default methods are a new way to add default implementation methods to Java interfaces.
- “Java interfaces can have implementations?!” Yes, Java 8 now has support for the concept, called virtual extension methods, better known as defender methods.
- Default methods have the effect of adding (extending) new behavior to interfaces without breaking compatibility.
- Default methods are not abstract methods but methods having implementation (code).
- For example, adding new abstract methods to Java interfaces can affect all implementation classes.
- Because of the strict contract of Java interfaces, the compiler forces the implementer to implement abstract methods, but classes that implement interfaces with default methods do not force the developer to implement those default methods.
- Rather, derived classes will acquire the behavior of the default method implementation.

# Default Methods in Interfaces

