Interface and package

Interface

- In Java, an interface is a collection of abstract methods. It is a way to achieve abstraction and multiple inheritance.
- An interface is declared using the interface keyword, and it can contain method signatures (without method bodies), constants, and nested types.
- Classes that implement an interface must provide concrete implementations for all the methods declared in the interface.

Interface

- ***Another way to achieve abstraction in Java, is with interfaces.***
- ***An interface is a completely "abstract class" that is used to group related methods with empty bodies:***

```
// interface
1 interface Animal {
2   public void animalSound();
// interface method (does not have a body)
3   public void run();
// interface method (does not have a body)
4 }
```

Interface

- *To access the interface methods, the interface must be "implemented" (kinda like inherited) by another class with the implements keyword (instead of extends).*
- *The body of the interface method is provided by the "implement" class:*

```
// Interface
interface Animal {
  public void animalSound(); // interface method (does not have a body)
  public void sleep(); // interface method (does not have a body)
}
// Pig "implements" the Animal interface
class Pig implements Animal {
  public void animalSound() {
    // The body of animalSound() is provided here
    System.out.println("The pig says: wee wee");
  }
  public void sleep() {
    // The body of sleep() is provided here
    System.out.println("Zzz");
  }
}
class Main {
  public static void main(String[] args) {
    Pig myPig = new Pig();  // Create a Pig object
    myPig.animalSound();
    myPig.sleep();
  }
}
```

Extending Interface

- *In Java, interfaces can extend other interfaces, forming an inheritance relationship between them.*
- *When one interface extends another, it inherits the abstract methods and constants of the parent interface.*
- *The extends keyword is used to indicate that one interface is an extension of another.*

```
// Parent interface
interface Animal {
    void makeSound();
}
// Child interface extending the parent interface
interface Mammal extends Animal {
    void giveBirth();
}
// Concrete class implementing the child interface
class Dog implements Mammal {
    @Override
    public void makeSound() {
        System.out.println("Woof!");
    }

    @Override
    public void giveBirth() {
        System.out.println("Dog giving birth");
    }
}
public class Main {
    public static void main(String[] args) {
        // Creating an object of the implementing class
        Dog myDog = new Dog();
        // Calling the implemented methods
        myDog.makeSound();
        myDog.giveBirth();
    }
}
```

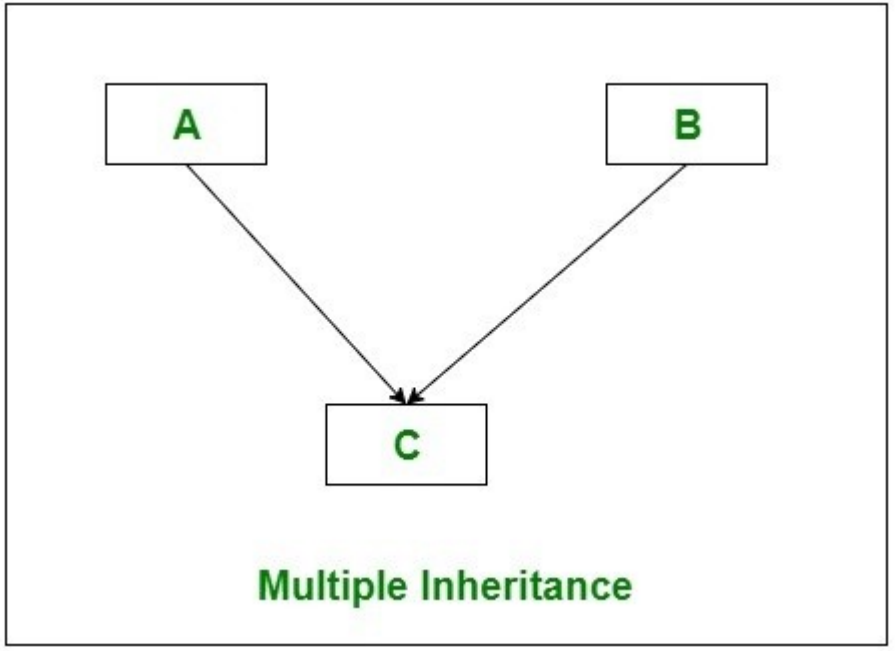
Interface

- Like abstract classes, interfaces cannot be used to create objects (in the example above, it is not possible to create an "Animal" object in the MyMainClass)
- Interface methods do not have a body - the body is provided by the "implement" class
- On implementation of an interface, you must override all of its methods
- Interface methods are by default abstract and public
- Interface attributes are by default public, static and final
- An interface cannot contain a constructor (as it cannot be used to create objects)
- **Why And When To Use Interfaces?**
  - To achieve security - hide certain details and only show the important details of an object (interface).
  - Java does not support "multiple inheritance" (a class can only inherit from one superclass). However, it can be achieved with interfaces, because the class can implement multiple interfaces.

Multiple Interfaces

To implement multiple interfaces, separate them with a comma:

```
interface FirstInterface {
  public void myMethod(); // interface method
}
interface SecondInterface {
  public void myOtherMethod(); // interface method
}
// DemoClass "implements" FirstInterface and SecondInterface
class DemoClass implements FirstInterface, SecondInterface {
  public void myMethod() {
    System.out.println("Some text..");
  }
  public void myOtherMethod() {
    System.out.println("Some other text...");
  }
}
class Main {
  public static void main(String[] args) {
    DemoClass myObj = new DemoClass();
    myObj.myMethod();
    myObj.myOtherMethod();
  }
}
```



Accessing Interface Variables

Accessing through the interface name:

```
interface MyInterface {
    int MY_CONSTANT = 42;
}
public class Main {
    public static void main(String[] args) {
        // Accessing the constant through the interface name
        int value = MyInterface.MY_CONSTANT;
        System.out.println("Constant value: " + value);
    }
}
```

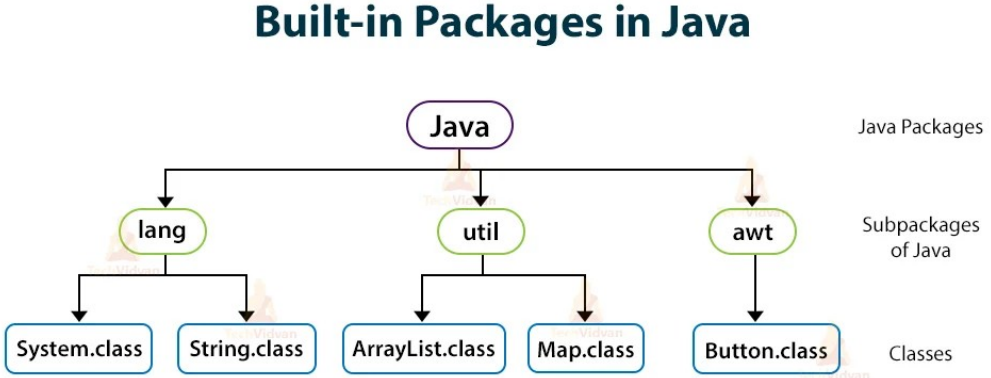
Accessing Interface Variables

Accessing through an implementing class:

```
interface MyInterface {
    int MY_CONSTANT = 42;
}
class MyClass implements MyInterface {
    void printConstant() {
        // Accessing the constant through the implementing class
        System.out.println("Constant value: " + MY_CONSTANT);
    }
}
public class Main {
    public static void main(String[] args) {
        // Creating an object of the implementing class
        MyClass myObject = new MyClass();
        // Accessing the constant through the object of the implementing class
        myObject.printConstant();
    }
}
```

Introduction to java Packages

- *In Java, a package is a way to organize related classes and interfaces into a single namespace.*
- *It helps in preventing naming conflicts and provides a modular structure to your code.*
- *A package can contain sub-packages, which further organize classes and interfaces within the broader package.*



Introduction to java Packages

- *A package in Java is used to group related classes. Think of it as a folder in a file directory.*
- *We use packages to avoid name conflicts, and to write a better maintainable code. Packages are divided into two categories:*
  - *Built-in Packages (packages from the Java API)*
  - *User-defined Packages (create your own packages)*

Introduction to java Packages

- Built-in Packages
  - The Java API is a library of prewritten classes, that are free to use, included in the Java Development Environment.
  - The library contains components for managing input, database programming, and much much more. The complete list can be found at Oracles website: https://docs.oracle.com/javase/8/docs/api/.
  - The library is divided into packages and classes.
  - Meaning you can either import a single class (along with its methods and attributes), or a whole package that contain all the classes that belong to the specified package.
  - To use a class or a package from the library, you need to use the ***import*** keyword:

Creating a Package and naming convention

- **Syntax:**
  - *import package.name.Class; // Import a single class*
  - *import package.name.*; // Import the whole package*
- ***Import a Class***
  - *If you find a class you want to use, for example, the Scanner class, which is used to get user input, write the following code:*
- *Example*
  - *import java.util.Scanner;*

Creating a Package and naming convention

- *In the example aside, java.util is a package, while Scanner is a class of the java.util package.*
- *To use the Scanner class, create an object of the class and use any of the available methods found in the Scanner class documentation.*
- *In our example, we will use the nextLine() method, which is used to read a complete line:*

```
1  import java.util.Scanner; // import the Scanner class
2  class Main {
3    public static void main(String[] args) {
4      Scanner myObj = new Scanner(System.in);
5      String userName;
6      // Enter username and press Enter
7      System.out.println("Enter username");
8      userName = myObj.nextLine();
9      System.out.println("Username is: " + userName);
10   }
11 }
```

User-defined Packages

- *User-defined Packages*
- *To create your own package, you need to understand that Java uses a file system directory to store them. Just like folders on your computer*

```
1 myapp
2 |-- src
3 |   |-- com
4 |       |-- example
5 |           |-- myapp
6 |               |-- YourClass1.java
7 |               |-- YourClass2.java
8 |               |-- Main.java
9 |-- out
```

Creating a Package and naming convention

- *Organizing your directory structure is crucial for maintaining a clean and structured codebase.*
- *Here is a suggested way to organize your Java project directory structure:*
- *Assuming your base package is com.example.myapp, you could structure your directories as follows:*

```
myapp
|-- src
|   |-- com
|       |-- example
|           |-- myapp
|               |-- YourClass1.java
|               |-- YourClass2.java
|               |-- Main.java
|-- out
```

Creating a Package and naming convention

- ***src**: This is where you keep your source code files.*
- *The directory structure under src mirrors your package structure.*
- *For example, the package com.example.myapp corresponds to the directory src/com/example/myapp.*
- *This is where you place your Java files.*
- ***out**: This is where you can store compiled Java classes. When you compile you*

```
1 myapp
2 |-- src
3 |   |-- com
4 |       |-- example
5 |           |-- myapp
6 |               |-- YourClass1.java
7 |               |-- YourClass2.java
8 |               |-- Main.java
9 |-- out
```

Creating a Package and naming convention

- *With this structure, when you compile your Java code, you might navigate to the root directory of your project and use a command like:*
- ***javac -d out src/com/example/myapp/*.java***
- *This command compiles all Java files in the src/com/example/myapp directory and places the compiled classes in the out directory.*
- To Execute: ***java -cp out Main***

```
1  myapp
2  |-- src
3  |   |-- com
4  |       |-- example
5  |           |-- myapp
6  |               |-- YourClass1.java
7  |               |-- YourClass2.java
8  |               |-- Main.java
9  |-- out
```

Creating a Package and naming convention

```java
// YourClass1.java
package com.example.myapp;
public class YourClass1 {
    public void display() {
        System.out.println("This is YourClass1");
    }
}
```

```java
// YourClass2.java
package com.example.myapp;
public class YourClass2 {
    public void display() {
        System.out.println("This is YourClass2");
    }
}
```

Creating a Package and naming convention

```
1  // Main.java
2  import com.example.myapp.YourClass1;
3  import com.example.myapp.YourClass2;
4  public class Main {
5      public static void main(String[] args) {
6          YourClass1 obj1 = new YourClass1();
7          YourClass2 obj2 = new YourClass2();
8          obj1.display(); // Output: This is
   YourClass1
9          obj2.display(); // Output: This is
   YourClass2
10     }
11 }
```

- *In this example:*
- *YourClass1 and YourClass2 are placed in the package com.example.myapp.*
- *The Main class imports these classes and creates objects to demonstrate the usage.*
- *When you run the Main class, it should output the respective messages from the display methods of YourClass1 and YourClass2.*

Assignment

- Create a Java program that demonstrates the concept of multiple inheritance using interfaces. Define an interface Drawable with a method draw(), and another interface Colorable with a method setColor(String color). Now, create a class named ColoredShape that implements both Drawable and Colorable interfaces. In the draw() method, print a message indicating the shape is being drawn, and in the setColor() method, set and display the color of the shape. Finally, create a main class named Main to instantiate an object of ColoredShape and invoke both methods.
- Design a simple Java program that uses the Scanner class from the java.util package for user input. Create a package named userinput and a class named InputProcessor inside it. In InputProcessor, prompt the user to enter two integers, perform their sum, and display the result. Finally, create a main class named Main to instantiate an object of InputProcessor and invoke the method for processing user input.