Java_OOP_2

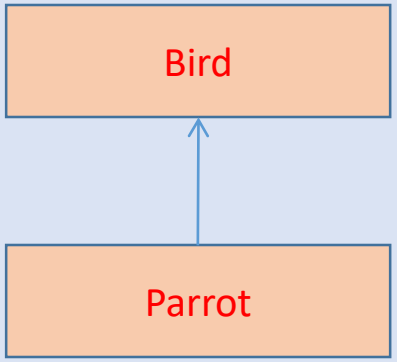
- **Casting in Inheritance**
- ***instanceof* operator**
- **Access control in Inheritance**
- **Covariant return types**
- **final keyword**
  - **final variable**
  - **final method**
  - **final class**
- **Abstract Classes**
- **Interfaces**
- **Comparable and Comparator Interfaces**

Inheritance: Casting objects

There are two types of casting:

1. Implicit casting:

The subclass type can be casted into super class type which is implicit. This process is also called as widening.



Ex. If Parrot class is a subclass of Bird class.

A reference to the instance of Parrot class can be type-casted to bird type, since it is perfectly valid to say that parrot is a bird.

```
Bird p = new Parrot(); // implicit cast
```

Inheritance: Casting objects

2. Explicit casting:

If the super class type is to be casted into subclass type , then explicit casting is required.

This process is also called as Narrowing.

Syntax:

SubClass p = (SubClassName) new SuperClassName();

Ex. To say that bird is a parrot, there needs to be a proof for that since a bird can be a sparrow or a wood-pecker or some other bird.

Parrot p = (Parrot) new Bird();

Note : Though we can cast super class instance into subclass type, an exception occurs at run time when we invoke the method, as shown below
Exception in thread "main" java.lang.ClassCastException: Bird cannot be cast to Parrot

instanceof operator

Java Project : InstanceOfProject

```
package com.deloitte.trg.service;

class A {

void showData(){

   System.out.println("showData() of class A");

}}

package com. deloitte.businesstier

class B extends A {

void show(){

   System.out.println("Hello");}

void showData(){

  System.out.println("showData() of class B");

} }
```

```
package com.deloitte.trg.servi
class C extends A {
void showData(){
  System.out.println("showData() of class C");
}
}
package com.deloitte.bussinesstier
public class D {
private String message;
public  void checkMessage(A  aObjRef){
  if( aObjRef instanceof  B){
        message="Received instance of class B";
     } else if(aObjRef instanceof  C){
        message="Received instance of class C";
    }
}
```

```
public String getMessage(){
        return message;
}
}
```

Note: We can make a logical test using *instanceof* operator. This can save you from a runtime error owing to an improper cast.

package com.deloitte.trg.ui;
Class Name: Tester

```
D d=new D();
d.checkMessage(new B());
System.out.println(d.getMessage());
d.checkMessage(new C());
System.out.println(d.getMessage());
```

Access Control in Inheritance

```
public class A {

 public  void  showMessage(){

        System.out.println("Hello");

}

}

public class B extends A {

@Override
 void showMessage(){
     System.out.println("Welcome");
  }

}
```

The overridden method of subclass can have broader access modifier but not narrower than the super class method.

```
@Override
public  void  showMessage(){
        System.out.println("Welcome");
}
```

```
public class A {

void  showMessage(){

        System.out.println("Hello");

}

}

public class B extends A {

 @Override
 public void showMessage(){
     System.out.println("Welcome");
  }

}
```

Covariant Return Type

An overriding method can also return a subtype of the type returned by the super class method. This is called *covariant return type*.

```
class A{
  .......
  A getInstance() //  return type is A
     {
       return new A();
     }
}
class B extends A{
  .......
  B getInstance() // return type is B
     {
       return new B();
     }
}
```

Constructors and methods throwing exceptions in super class.

- Handling in sub classes.

Constructors & Methods throwing exceptions

```
public class SuperClass {
public SuperClass() throws java.io.IOException{
 System.out.println("Hi");
}
public void show() throws RuntimeException{
 System.out.println("I'm in super class show() method");
} }
```

If a **constructor method** in super class throws an **checked exception**, then the sub class constructor method should throw *same type of exception or its super type.*

If a **constructor method** in super class throws an **unchecked exception**, then the constructor method in the sub class has following options:

a. Can throw same type of exception
b. Can throw its super type exception
c. Can ignore throwing an exception

```
public class SubClass extends SuperClass {
  // or throws java.io.Exception
public SubClass() throws java.io.IOException
            System.out.println("Welcome");
   }
// valid , we can ignore throwing an exception
public void show() throws ArithmeticException{
    super.show();
    System.out.println("I'm in sub class show() method");
}
}
```

```
public static void main(String[] args) throws Exception{
   new SubClass().show();
 }
```

If a instance method in super class throws an **unchecked exception**, then the overridden method in the sub class has following options:

a. Can throw same type of exception
b. Can throw its sub type exception
c. Can ignore throwing an exception

If a instance method in super class throws an **checked exception**, then the overridden method in the sub class has to throw **same exception type or its sub-type**.

Constructors & Methods throwing exceptions

```
class Parent{
protected int a;
Parent() throws Exception{
        System.out.println("Parent class");
}
public void getData() {
System.out.println("Hi");
}
public void showData(){
  System.out.println("a= " + a);
}}
```

```
class Child extends Parent{
Child() throws Exception{
        super();
        System.out.println("Child class");
}
@Override
public void getData() throws IOException{
        BufferedReader bufferedReader =
         new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Enter a number ");
        int a = Integer.parseInt(br.readLine());
}}
```

Note: public void getData() throws RuntimeException is valid

If *getData()* of super class does not declare *throws IOException* and if the overridden method in subclass declares, it leads to compile error.

The following exception is throw:

Exception IOException is not compatible with throws clause in Parent.getData()

final keyword

final keyword *can be applied to* member variables, *local variables*, methods or classes in *Java.*

Use final keyword for declaring:

- final variables
  - final member Variables
  - final local Variables
- final methods
- final classes

final member variables

- Hold values that do not change during the execution of the application. Final variables are equivalent to constants.

Syntax:

public static final datatype variableName=value;

final local variables

- Are declared inside a method.

Syntax:

final datatype variableName=value;

Ex.

```
public void method1(){
    final int MAX_VAL=100;
            OR
    final int MAX_VAL;
    MAX_Val=100;
}
```

Note: one-time assignment of local final variable is allowed.

final methods and final classes

final methods can not be overridden.

Syntax:

```
access_modifier final return_type methodName() {
  //code
}
```

- final , private, static methods and overloaded methods come under *static binding.*
- Overridden Instance methods fall under *dynamic binding.*

Final classes cannot be sub-classes.

When a class is declared with the final modifier, it means that it cannot be extended or sub classed.

In addition, all methods of a final class are themselves implicitly final and cannot be overridden.

Syntax:

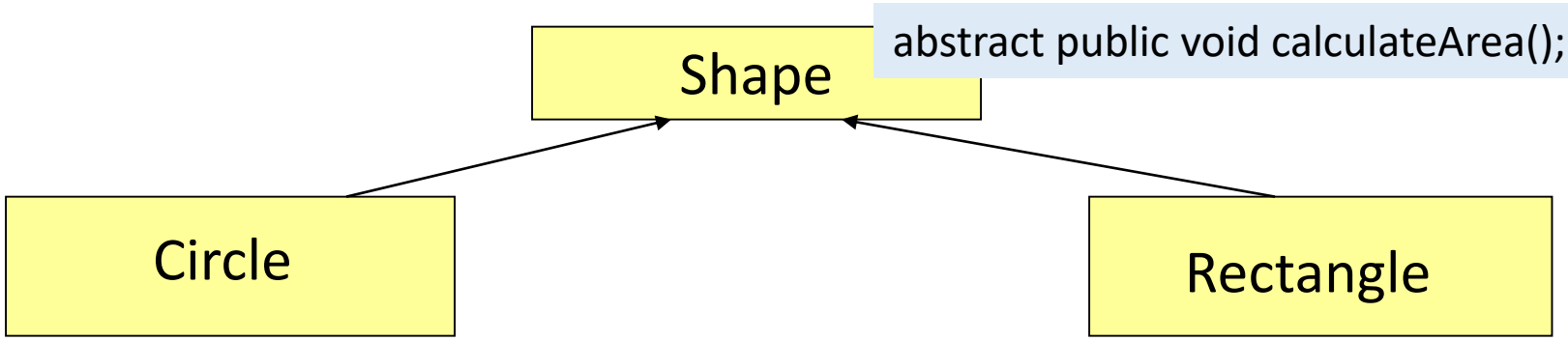
```
access_modifier final class ClassName{
      .....
}
```

java.lang.System is an example of a *final class*.

Abstract Classes

Abstract Class

- Consider a scenario where you consider a generalized class Shape which has two subclasses Circle and Rectangle
- Circle and Rectangle have their own way to calculate area
- So, Shape class cannot decide how to calculate the area
  - it only provides the guideline to the child classes that such a method should be implemented in them
- *public void calculateArea()* method has been declared abstract in the super class i.e. the method signature has been provided but the ***implementation has been deferred.***




```
abstract class Shape{
        //Class definition
}
```

Abstract Class

- A Java abstract class is a class **which cannot be instantiated**, meaning you cannot create new instances of an abstract class.
- The purpose of an abstract class is to function as a base for **subclasses**.
- The ***abstract*** keyword can be applied for methods and for classes.
- An ***abstract method*** signifies it has no body (implementation), only declaration of the method followed by a semicolon

  Ex. *abstract public double calculateArea();*

- If a class has one or more abstract methods declared inside it, the class must be declared abstract.
- Reference of an abstract class can point to objects of its sub-classes thereby achieving run-time polymorphism


```
Shape s = new Circle();
```

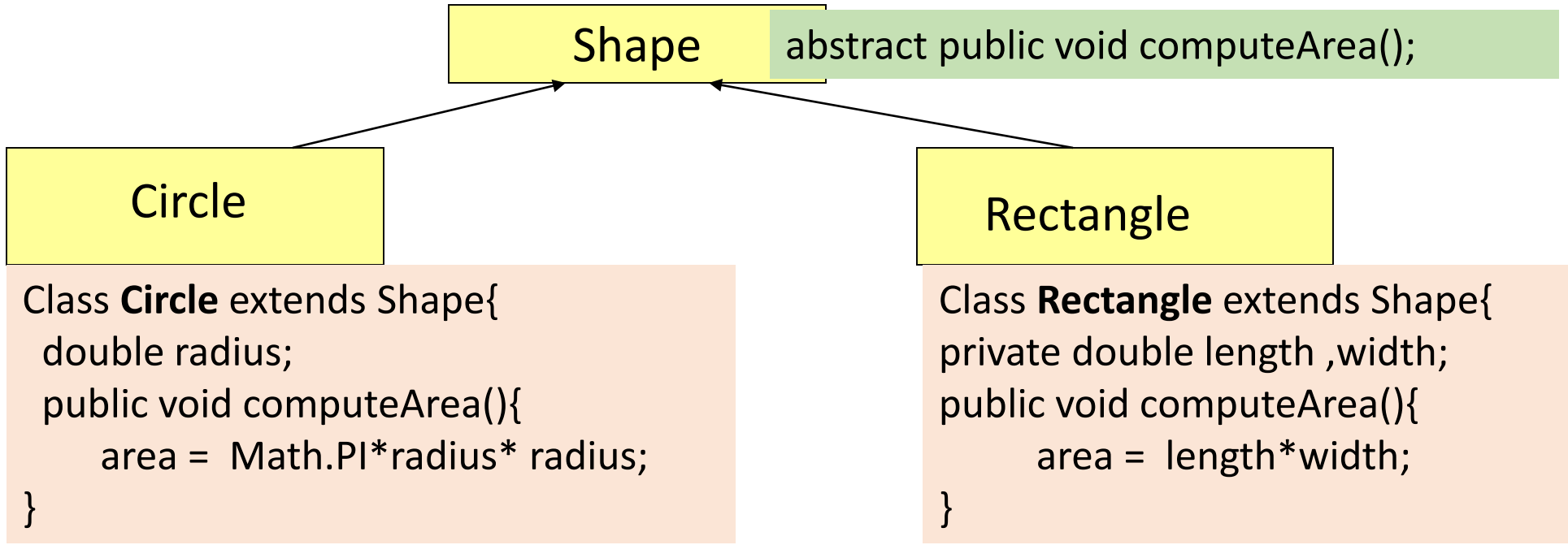
Abstract Class

```
abstract class Shape{
      abstract public void calculateArea();
      public void setColor(){
            //code to color the shape
      }
}
```

Note:

- An abstract class may also have concrete (complete) methods and fields.
- **For design purpose, a class can be declared abstract even if it does not contain any abstract methods**

Abstract Class



~~Shape shape = new Shape();~~

Run-time polymorphism

```
Shape shape; // reference variable
shape = new Circle();
shape.computeArea();

shape = new Rectangle();
shape.computeArea();
```

Invokes computeArea() of Circle class

Invokes computeArea() of Rectangle class

Abstract Class

Summary

- Any class with an abstract method is automatically abstract itself, and must be declared as such.

- A class may be declared abstract even if it has no abstract methods. This prevents it from being instantiated.

- An abstract class cannot be instantiated.

- A subclass of an abstract class can be instantiated if it overrides each of the abstract methods of its super class and provides an implementation (i.e., a method body) for all of them.

- If a subclass of an abstract class does not implement all of the abstract methods it inherits, that subclass is itself abstract.

Java OOP-2

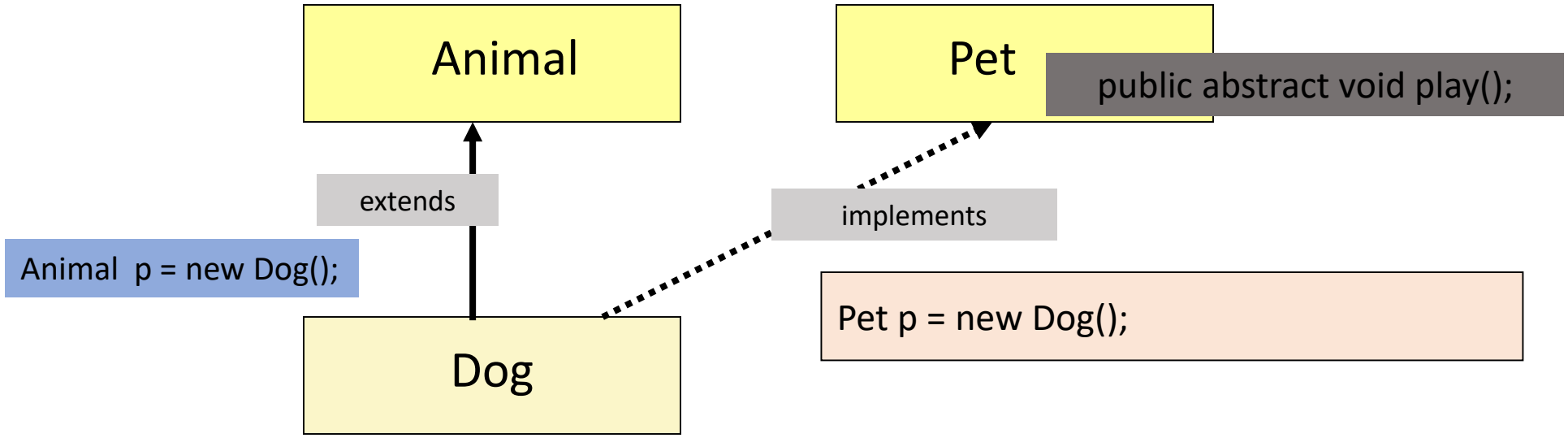
Interfaces

Interface

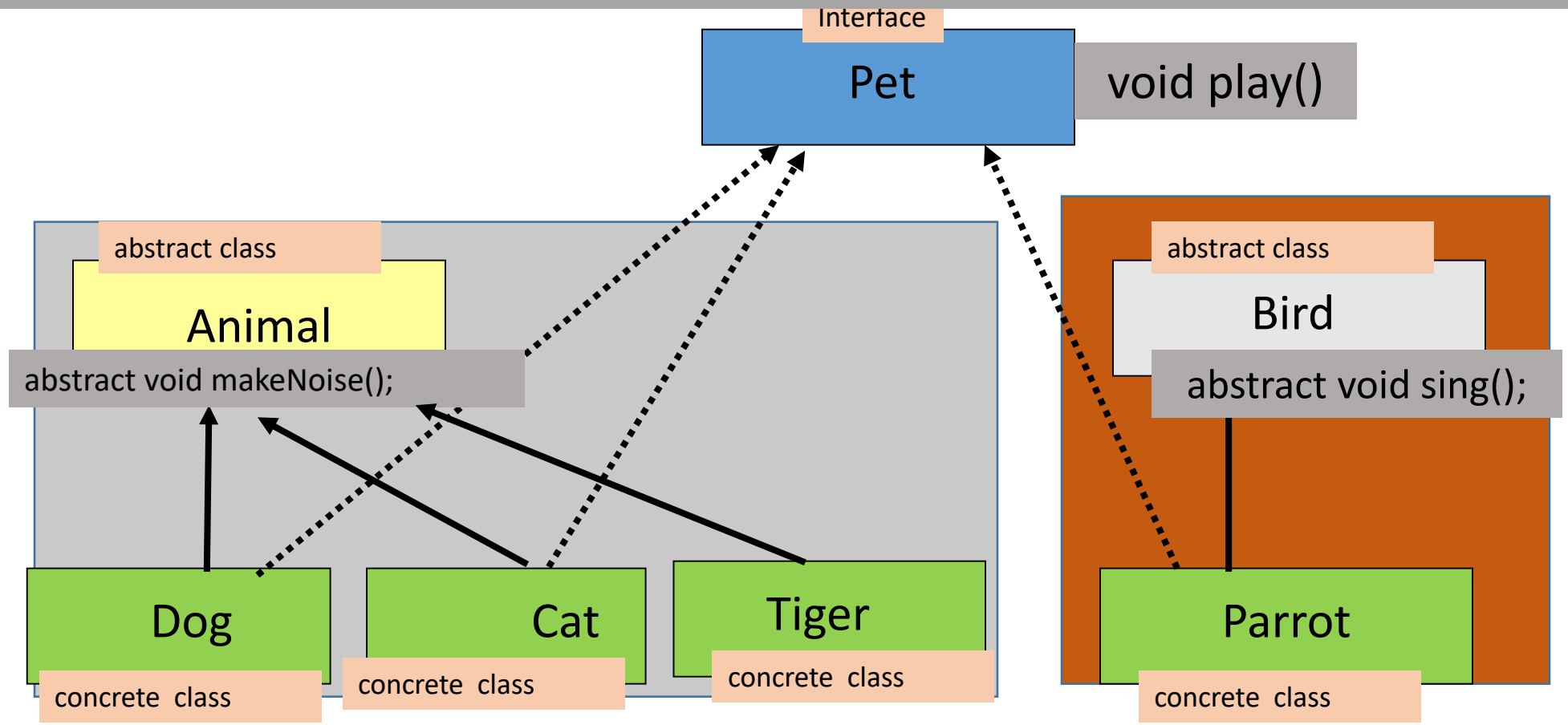
- Let us consider a design option: class Dog has been designed which is a sub-class of Animal
- Dog is also a Pet and it needs the behaviors of a pet
- So, should we place the functionalities of a Pet in class Dog?
- Then what happens when we design a class Cat which extends Animal and is also a Pet ?
- Then what happens when we design a class Tiger which extends Animal and is not a Pet

Interface

- Interface can rescue us
- Interface is similar to an abstract class that contains only abstract methods
- The functionalities of Pet can be placed in an interface
- So class Dog now ***extends*** class Animal and ***implements*** Pet



Interface




```
Pet  pet[] = new Pet[3];

pet[0] = new Dog();
pet[1] = new Parrot();
pet[2] = new Cat();
```

```
pet[0].makeNoise();
pet[1].sing();
pet[2].play();
```

```
Animal animal[] = new Animal[3];
animal[0] = new Dog(); animal[1] = new Tiger();
animal2] = new Cat();

animal[0].makeNoise();
animal[1].play();
animal[2].play();
```

Interface

Interfaces cannot be instantiated—they can only be *implemented* by classes or *extended* by other interfaces.

An interface declaration

[*modifiers*] interface *InterfaceName* [extends Interface1, …]
{
// declaring methods
[public abstract] returnType methodName1(*arguments*);

// defining constants
[public static final] type propertyName = value;
}

```
public interface Pet{
        void beFriendly();
        void play();
}
```

An interface can contain only **public abstract instance methods** and **public static final variables**

A class can extend only one other class, an interface can extend any number of interfaces.

Interface

- All methods in an interface are implicitly ***public*** and ***abstract***
- The class which implements the interface *(called as implementation class*) needs to provide functionality for the methods declared in the interface.
- The implementation class needs to be declared abstract even if one method of the interface is left undefined

Interface

To declare a class that implements an interface, include an *implement*s clause in the class declaration.

A class can implement more than one interface, so the *implements* keyword is followed by a comma-separated list of the interfaces implemented by the class.

The *implements* clause must follow the *extends* clause. (Reversing leads to compile error)

[*modifiers*] class *ClassName* [extends ClassName] implements *InterfaceName1[, InterfaceName2* ..] {

```
 // any fields

// implement methods declared in the interface
 public returnType methodName1(arguments)
 {
  executable code
 }

//other methods

 }
```

```
class Dog extends Animal implements Pet{
    public void beFriendly(){
        //functionality
    }
    public void play(){
        //functionality
    }
        //other functions
}
```

Loose Coupling

One of the primary goals in Object Oriented Design is to design the system in such away that various objects are loosely coupled with each other to increase the flexibility of the overall system.

To understand loose coupling, consider the following example. Let's say your company want to build a *search engine that uses web tool* to search for resources over the web.

Java Project: InterfaceLooseCoupling

```
public interface SearchTool{
  public abstract void search();
}
```

```
public class WebTool implements SearchTool{
public void search() {
System.out.println("Found 10 results in Web");
}
}
```

```
public class ImageTool implements SearchTool{
public void search() {
System.out.println("Found 10 image results");
}
}
```

```
public class SearchEngine {
public void performSearch(SearchTool tool) {
   tool.search();
}
}
```

```
SearchEngine engine = new SearchEngine();
engine.performSearch(new WebTool());
engine.performSearch(new ImageTool());
```

In most of the real world applications, the arguments of a method will always be a *parent class references or interface references* to achieve loose coupling.

Marker/Tagged interface

Marker or Tagged Interface

- An empty interface is called as marker or tagged interface.
- A class can implement this interface to provide additional information about itself.
- The *java.lang.Cloneable* interface is an example of marker interface.
- It defines no methods, but serves simply to identify the class as one that will allow its internal state to be cloned by the clone() method of the Object class.
- *java.io.Serializable* is another such marker interface.

```
public interface InterfaceName{
}
```

Functional interface

A functional interface is an interface that contains only one abstract method. They can have only one functionality to exhibit.

From Java 8 onwards, lambda expressions can be used to represent the instance of a functional interface.

Runnable, ***ActionListener***, ***Comparable*** , ***Comparator*** are some of the examples of functional interfaces.

Java OOP-2

Comparable and Comparator interfaces

Comparable and Comparator Interfaces

Comparable and *Comparator* are two interfaces which are used to *implement sorting in Java*.

It is often required to sort objects stored in any collection classes like ArrayList, HashSet or in an Array and that time we need to implement either *compareTo()* method *of java.lang.Comparable* interface or *compare()* method of *java.util.Comparator.*

Note: **In Java API String, Date and wrapper classes implement Comparable interface.**

1. Comparable interface of java.lang package has the following method:

public int compareTo(Object o) : returns a negative integer, zero, or a positive integer.

2. Comparator interface of java.util package has the following method:

public int compare (Object o1, Object o2) : returns a negative integer, zero, or a positive integer .

Comparable and Comparator Interfaces

compareTo() method of **Comparable** interface in Java is used to implement natural ordering of object.

If a class implements *Comparable interface* in Java then collection of that objects either in *List* or *Array* can be sorted automatically by using **Collections.sort()** or **Arrays.sort()** methods.

So in Summary, To sort objects based on *natural order,* such as employeeID, admissionCode, accountNumber etc implement Comparable Interface and to sort on some other attribute of an object such as employee name , title of a book etc. then implement Comparator Interface .

Comparable and Comparator Interfaces

```
import java.util.Arrays;
import com.deloitte.trg.service.Student;

public class StudentTester {
public static void main(String[] args) {
Student student1=new Student(1105,"Anil Kumar",89);
Student student2=new Student(1103,"Madhav",80);
Student student3=new Student(1102,"Laxmi",90);
Student student4=new Student(1101,"Venu Gopal",75);

Student [] students={student1,student2,student3,student4};

for(Student student: students){
System.out.println(student);
}

Arrays.sort(students);

for(Student student: students){
System.out.println(student);
}
}}
```

```
public class Student {
    int admissionCode;
    String studentName;
    public int marks;
    ……..
}
```

Runtime error

Solution : We have to implement either Comparable or Comparator interface.

```
Exception in thread "main" java.lang.ClassCastException:
com.deloitte.businesstier.Student cannot be cast to
java.lang.Comparable
at
java.util.ComparableTimSort.countRunAndMakeAscending(Unknown
Source)
at java.util.ComparableTimSort.sort(Unknown Source)
at java.util.Arrays.sort(Unknown Source)
at
com.deloitte.presentationtier.StudentTester.main(StudentTester
.java:21)
```

Implementing Comparable Interface

```
public class Student implements Comparable<Student>{
int admissionCode;
String studentName;
public int marks;
.....
@Override
public String toString() {
return "Student [admissionCode=" + admissionCode + ", studentName=" + studentName + ", marks=" + marks + "]";
}
@Override
public int compareTo(Student that) {
if( this.admissionCode < that.admissionCode ){
        return -1;
}
else if ( this.admissionCode > that.admissionCode ){
        return 1;
}
        return 0;
}
}
```

Generics

Re-run the StudentTester class

Implementing Comparator interface

To perform sorting based on other field , create a separate class that implements Comparator class.

Let us create a class that implements Comparator interface to sort Student objects based on student names.

```
import java.util.Comparator;

public class StudentNameComparator implements Comparator<Student>{

@Override
public int compare(Student o1, Student o2) {
      return o1.getStudentName().compareTo(o2.getStudentName());
}
}
```

```
public class StudentTester {
      public static void main(String[] args) {
      …..
      Arrays.sort(students, new StudentNameComparator());
      …..
}
}
```

We can create implementation classes for other fields too.

Thank You!