Core Java

Inheritance

Types of inheritances

- Single inheritance

```
class A {
    // ...
}
class B extends A {
    // ...
}
```

- Multiple inheritance

```
class A {
    // ...
}
class B {
    // ...
}
class C extends A, B // not allowed in Java
{
    // ...
}
```

```
interface A {
    // ...
}
interface B {
    // ...
}
class C implements A, B // allowed in Java
{
    // ...
}
```

- Hierarchial inheritance

```
class A {
    // ...
}
```

```
class B extends A {
    // ...
}
class C extends A {
    // ...
}
```

- Multi-level inheritance

```
class A {
    // ...
}
class B extends A {
    // ...
}
class C extends B {
    // ...
}
```

- Hybrid inheritance: Any combination of above types

Up-casting & Down-casting

- Up-casting: Assigning sub-class reference to a super-class reference.
    - Sub-class "is a" Super-class, so no explicit casting is required.
    - Using such super-class reference, super-class methods overridden into sub-class can also be called.

```
Employee e = new Employee();
Person p = e; // up-casting
p.setName("Nilesh");    // okay - calls Person.setName().
p.setSalary(30000.0);   // error
p.display();    // calls overridden Employee.display().
```

- Down-casting: Assigning super-class reference to sub-class reference.
    - Every super-class is not necessarily a sub-class, so explicit casting is required.

```
Person p1 = new Employee();
Employee e1 = (Employee)p1; // down-casting - okay - Employee reference will
point to Employee object
```

```
Person p2 = new Person();
Employee e2 = (Employee)p2; // down-casting - ClassCastException - Employee
```

```
reference will point to Person object
```

instanceof operator

- Java's instanceof operator checks if given reference points to the object of given type (or its sub-class) or not. Its result is boolean.

```
Person p = new Employee();
boolean flag = p instanceof Employee; // true
```

```
Person p = new Manager();
boolean flag = p instanceof Employee; // true
```

- Typically "instanceof" operator is used for type-checking before down-casting.

```
Person p = new SomeClass();
if(p instanceof Employee) {
    Employee e = (Employee)p;
    System.out.println("Salary: " + e.getSalary());
}
```

Polymorphism

- Poly=Many, Morphism=Forms
- Polymorphism is taking many forms.
- OOP has two types of Polymorphism
    - Compile-time Polymorphism / Static Polymorphism
        - Implemented by "Method overloading".
        - Compiler can identify which method to be called at compile time depending on types of arguments. This is also referred as "Early binding".
    - Run-time Polymorphism / Dynamic Polymorphism
        - Implemented by "Method overriding".
        - The method to be called is decided at runtime depending on type of object. This is also referred as "Late binding" or "Dynamic method dispatch".
    - Process of calling method of sub class using reference of super class reference is called as dynamic method dispatch. ###Access Modifier
    - private (lowest)
    - default
    - protected
    - public (highest)

Method overriding

- Redefining a super-class method in sub-class with exactly same signature is called as "Method overriding".
- Programmer should override a method in sub-class in one of the following scenarios
  - Super-class has not provided method implementation at all (abstract method).
  - Super-class has provided partial method implementation and sub-class needs additional code. Here sub-class implementation may call super-class method (using super keyword).
  - Sub-class needs different implementation than that of super-class method implementation.
- Rules of method overriding in Java
  - Each method in Java can be overridden unless it is private, static or final.
  - Sub-class method must have same or wider access modifier than super-class method.

```
class SuperClass {
    protected Number calculate(Integer i, Float f) {
        // ...
    }
}
class SubClass extends SuperClass {
    /*protected or*/ public Number calculate(Integer i, Float f) {
        // ...
    }
}
```

  - Arguments of sub-class method must be same as of super-class method. The return-type of sub-class method can be same or sub-class of the super-class's method's return-type. This is called as "covarient" return-type.

```
class SuperClass {
    public Number calculate(Integer i, Float f) {
        // ...
    }
}
class SubClass extends SuperClass {
    // Double is inherited from Number (i.e. return-type of super-class
method)
    public Double calculate(Integer i, Float f) {
        // ...
    }
}
```

  - Checked exception list in sub-class method should be same or subset of exception list in super-class method.

```
class SuperClass {
    public void testMethod() throws IOException, SQLException {
        // ...
    }
```

```
}
class SubClass extends SuperClass {
    public void testMethod() throws IOException {
        // ...
    }
}
```

- If these rules are not followed, compiler raises error or compiler treats sub-class method as a new method.

```
class A {
    void method1() {
    }
    void method2() {
    }
}
```

```
class B extends A {
    void method1() {
        // overridden
    }
    void method2(int x) {
        // treated as new method
    }
}
```

```
// In main()
A obj = new B();
obj.method1();  // B.method1() -- overridden
obj.method2();  // A.method2() -- method2() is not overridden
```

- Java 5.0 added @Override annotation (on sub-class method) informs compiler that programmer is intending to override the method from the super-class.
- @Override checks if sub-class method is compatible with corresponding super-class method or not (as per rules). If not compatible, it raise compile time error.
- Note that, @Override is not compulsory to override the method. But it is good practice as it improves readability and reduces human errors.

final keyword

final variables/fields

- Cannot be modified once initialized.
- Refer earlier notes.

final method

- If implementation of a super-class method is logically complete, then the method should be declared as final.
- Such final methods cannot be overridden in sub-class. Compiler raise error, if overridden.
- But final methods are inherited into sub-class i.e. The super-class final methods can be invoked in sub-class object (if accessible).

final class

- If implementation of a super-class is logically complete, then the class should be declared as final.
- The final class cannot be extended into a sub-class. Compiler raise error, if inherited.
- Effectively all methods in final class are final methods.
- Examples of final classes
    - java.lang.Integer (and all wrapper classes)
    - java.lang.String
    - java.lang.System

Object class

- Non-final and non-abstract class declared in java.lang package.
- In java, all the classes (not interfaces) are directly or indirectly extended from Object class.
- In other words, Object class is ultimate base class/super class.
- Object class is not inherited from any class or implement any interface.
- It has a default constructor.
    - Object o = new Object();

Object class methods (read docs)

- Parameter less constructor
    - public Object();
- Returns string representation of object state
    - public String toString();
- Comparing current object with another object
    - public boolean equals(Object);
- Used while storing object into set or map collections
    - public native int hashCode();
- Create shallow copy of the object
    - protected native Object clone() throws CloneNotSupportedException;
- Called by garbage collector (like C++ destructor)
    - protected void finalize() throws Throwable;
- Get metadata about the class
    - public final native Class<?> getClass();
- For thread synchronization
    - public final native void notify();
    - public final native void notifyAll();
    - public final void wait() throws InterruptedException;
    - public final native void wait(long) throws InterruptedException;

    - public final void wait(long, int) throws InterruptedException;

toString() method

- Non-final method of java.lang.Object class.
    - public String toString();
- Definition of Object.toString():

```
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

- To return state of Java instance in String form, programmer should override toString() method.
- The result in toString() method should be a concise, informative, and human-readable.
- It is recommended that all subclasses override this method.
- Example:

```
class Person {
    // ...
    @Override
    public String toString() {
        return "Name=" + this.name + ", Age=" + this.age;
    }
}
```

Inheritance vs Association

- Inheritance: is-a relation
    - Book is-a Product
    - Album is-a Product
    - Labor is-a Employee
    - Employee is-a Person
    - Batter is-a Player
    - ...
- Association: has-a relation
    - Employee has-a joining Date
    - Person has-a birth Date
    - Cart has Products
    - Bank has Accounts
    - ...

abstract keyword

- In Java, abstract keyword is used for
    - abstract method
    - abstract class

Fragile base class problem

- If changes are done in super-class methods (signatures), then it is necessary to modify and recompile all its sub-classes. This is called as "Fragile base class problem".
- This can be overcomed by using interfaces.

```
class A{
    public void print( ){
        //System.out.print("Hello,");
        System.out.print("Good Morning,");
    }
}
class B extends A{
    @Override
    public void print( ){
        super.print();
        System.out.println("Have a nice day!!");
    }
}
class C extends A{
    @Override
    public void print( ){
        super.print();
        System.out.println("Good day!!");
    }
}
class Program{
    public static void main(String[] args) {
        A a = null;
        a = new B();    a.print();  //Good Morning,,Have a nice day!!
        a = new C();    a.print();  //Good Morning,,Good day!!
    }
}
```

Interface (Java 7 or Earlier)

- Interfaces are used to define standards/specifications. A standard/specification is set of rules.
- Interfaces are immutable i.e. once published interface should not be modified.
- Interfaces contains only method declarations. All methods in an interface are by default abstract and public.
- They define a "contract" that is must be followed/implemented by each sub-class.

```
interface Displayable {
    public abstract void display();
}
```

```
interface Acceptable {
    abstract void accept(Scanner sc);
}
```

```
interface Shape {
    double calcArea();
    double calcPeri();
}
```

- Interfaces enables loose coupling between the classes i.e. a class need not to be tied up with another class implementation.
- Interfaces cannot be instantiated, they can only be implemented by classes or extended by other interfaces.
- Java 7 interface can only contain public abstract methods and static final fields (constants). They cannot have non-static fields, non-static methods, and constructors.
- Examples:
    - java.io.Closeable / java.io.AutoCloseable
    - java.lang.Runnable
    - java.util.Collection, java.util.List, java.util.Set, ...
- Example 1: Multiple interface inheritance is allowed.

```
interface Displayable {
    void display();
}
interface Acceptable {
    void accept();
}

class Person implements Acceptable, Displayable {
    // ...
    public void accept() {
        // ...
    }
    public void display() {
        // ...
    }
}
```

- Example 2: Interfaces can have public static final fields.

```
interface Shape {
    /*public static final*/ double PI = 3.142;

    /*public abstract*/ double calcArea();
    /*public abstract*/ double calcPeri();
```

```
}

class Circle implements Shape {
    private double radius;
    // ...
    public double calcArea() {
        return PI * this.radius * this.radius;
    }
    public double calcPeri() {
        return 2 * Shape.PI * this.radius;
    }
}
```

- Example 3: If two interfaces have same method, then it is implemented only once in sub-class.

```
interface Displayable {
    void print();
}
interface Showable {
    void print();
}
class MyClass implements Displayable, Showable {
    // ...
    public void print() {
        // ...
    }
}
class Program {
    public static void main(String[] args) {
        Displayable d = new MyClass();
        d.print();
        Showable s = new MyClass();
        s.print();
        MyClass m = new MyClass();
        m.print();
    }
}
```

Types of inheritance in OOPS

- Interface inheritance
  - Single inheritance [Allowed in Java]
  - Multiple inheritance [Allowed in Java]
  - Hierarchical inheritance [Allowed in Java]
  - Multilevel inheritance [Allowed in Java]
- Implementation inheritance
  - Single inheritance [Allowed in Java]
  - Multiple inheritance [Not Allowed in Java]
  - Hierarchical inheritance [Allowed in Java]

    - Multilevel inheritance [Allowed in Java]

- Interface syntax
    - Interface : I1, I2, I3
    - Class : C1, C2, C3
    - class C1 implements I1 // okay
    - class C1 implements I1, I2 // okay
    - interface I2 implements I1 // error
    - interface I2 extends I1 // okay
    - interface I3 extends I1, I2 // okay
    - class C2 implements C1 // error
    - class C2 extends C1 // okay
    - class C3 extends C1, C2 // error
    - interface I1 extends C1 // error
    - interface I1 implements C1 // error
    - class C2 implements I1, I2 extends C1 // error
    - class C2 extends C1 implements I1,I2 // okay

abstract method

- If implementation of a method in super-class is not possible/incomplete, then method is declared as abstract.
- Abstract method does not have definition/implementation.

```
// Employee class
abstract double calcTotalSalary();
```

- If class contains one or more abstract methods, then class must be declared as abstract. Otherwise compiler raise an error.
- The super-class abstract methods must be overridden in sub-class; otherwise sub-class should also be marked abstract.
- The abstract methods are forced to be implemented in sub-class. It ensures that sub-class will have corresponding functionality.
- The abstract method cannot be private, final, or static.
- Example: abstract methods declared in Number class are:
    - abstract int intValue();
    - abstract float floatValue();
    - abstract double doubleValue();
    - abstract long longValue();

abstract class

- If implementation of a class is logically incomplete, then the class should be declared abstract.
- If class contains one or more abstract methods, then class must be declared as abstract.
- An abstract class can have zero or more abstract methods.
- Abstract class object cannot be created; however its reference can be created.

- Abstract class can have fields, methods, and constructor.
- Its constructor is called when sub-class object is created and initializes its (abstract class) fields.
- If object of a class is not logical (corresponds to real-world entity), then class can be declared as abstract.
- Example:
    - java.lang.Number
    - java.lang.Enum

class vs abstract class vs interface

- class
    - Has fields, constructors, and methods
    - Can be used standalone -- create objects and invoke methods
    - Reused in sub-classes -- inheritance
    - Can invoke overridden methods in sub-class using super-class reference -- runtime polymorphism
- abstract class
    - Has fields, constructors, and methods
    - Cannot be used independently -- can't create object
    - Reused in sub-classes -- inheritance -- Inherited into sub-class and must override abstract methods
    - Can invoke overridden methods in sub-class using super-class reference -- runtime polymorphism
- interface
    - Has only method declarations
    - Cannot be used independently -- can't create object
    - Doesn't contain anything for reusing (except static final fields)
    - Used as contract/specification -- Inherited into sub-class and must override all methods
    - Can invoke overridden methods in sub-class using super-class reference -- runtime polymorphism
    - Java support multiple interface inheritance

What is the difference between abstract class and interface? / When we should use abstract class and interface?

Abstract class

```
abstract class Shape{
    public abstract void calculateArea( );
}
class Rectangle extends Shape{
    @Override
    public void calculateArea(){
        //TODO
    }
}
class Circle extends Shape{
    @Override
```

```
    public void calculateArea(){
        //TODO
    }
}
class Triangle extends Shape{
    @Override
    public void calculateArea(){
        //TODO
    }
}
```

- If "is-a" relationship is exist between super type & sub type and if we want to maintain same method signature/design in all the sub clases then we should declare super type abstract.

```
Shape[] arr = new Shape[ 3 ];
arr[ 0 ] = new Rectangle();
arr[ 1 ] = new Circle();
arr[ 2 ]  = new Triangle();
```

- Using abstract class, we can group instances of related type together.
- Abstract class can extend only one abstract class / concrete class. In other words, using abstract class, we can not achieve multiple inheritance.
- We can define constructor inside abstract class.
- Abstract class may / may not contain abstract method.
- In General, if state is involved in super type then super type should be abstract class.

Interface

```
interface Printable{
    void printRecord( );
}
class Complex implements Printable{
    @Override
    public void printRecord(){
        System.out.println("Print complex number");
    }
}
class Point implements Printable{
    @Override
    public void printRecord(){
        System.out.println("Print point");
    }
}
class Date implements Printable{
    @Override
    public void printRecord(){
        System.out.println("Print date");
```

```
    }
}
```

- If "is-a" relationship is not exist(can-do relationship is exist) between super type & sub type and if we want to maintain same method design in all the sub classes then we should declare super type interface.

```
Printable[] arr = new Printable[ 3 ];
arr[ 0 ] = new Complex();
arr[ 1 ] = new Point();
arr[ 2 ] = new Date();
```

- Using interface, we can group instances of unrelated type together.
- Interface can extend more than one interfaces. In other words, using interface, we can achive multiple inheritance.
- We can not define constructor inside interface.
- Interface methods are by default abstract.
- In General, if state is not involved in super type then super type should be interface.

equals() method

- Non-final method of java.lang.Object class.
    - public boolean equals(Object other);
- Definition of Object.equals():

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

- To compare the object contents/state, programmer should override equals() method.
- This equals() must have following properties:
    - Reflexive: for any non-null reference value x, x.equals(x) should return true.
    - Symmetric: for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
    - Transitive: for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
    - Consistent: for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.
    - For any non-null reference value x, x.equals(null) should return false.
- Example:

```
class Employee {
    // ...
```

```
    @Override
    public boolean equals(Object obj) {
        if(obj == null)
            return false;

        if(this == obj)
            return true;

        if(! (obj instanceof Employee))
            return false;

        Employee other = (Employee) obj;
        if(this.id == other.id)
            return true;
        return false;
    }
}
```

Marker interfaces

- Interface that doesn't contain any method declaration is called as "Marker interface".
- These interfaces are used to mark or tag certain functionalities/features in implemented class. In other words, they associate some information (metadata) with the class.
- Marker interfaces are used to check if a feature is enabled/allowed for the class.
- Java has a few pre-defined marker interfaces. e.g. Serializable, Cloneable, etc.
    - java.io.Serializable -- Allows JVM to convert object state into sequence of bytes.
    - java.lang.Cloneable -- Allows JVM to create copy of the class object.

Cloneable interface

- Enable creating copy/clone of the object.
- If a class is Cloneable, Object.clone() method creates a shallow copy of the object. If class is not Cloneable, Object.clone() throws CloneNotSupportedException.
- A class should implement Cloneable and override clone() to create a deep/shallow copy of the object.

```
class Date implements Cloneable {
    private int day, month, year;
    // ...
    // shallow copy
    public Object clone() throws CloneNotSupportedException {
        Date temp = (Date)super.clone();
        return temp;
    }
}
```