- If has-a relationship is exist between the type then we should use association.
- In case of association, we should declare instance of a class as a field inside another class.
- Example: Employee has a join date.

```
class Date{
    //TODO
}
class Employee{
    private Date joinDate;  //Association
}
class Program{
    public static void main(String[] args) {
        Employee emp = new Employee( );
    }
}
```

- In Java, instance do not get space inside another instance. Rather reference get space inside instance. It means that association is achived using reference variable.
- When instance is outside instance then it represents associaition and associaition represents loose coupling.

- If "is-a" relationship is exist between the type then we should use inheritance.
- Inheritance is also called as generalization.
- Consider following example
    - Manager is a Employee
    - Circle is a Shape
    - Saving account is a account.
    - Production department is a department.
- Journey from generalization to specialization is called as inheritance.
- Consider Example: Employee is a Person.

```
class Person{   //Parent / Super class
    //TODO
}
class Employee extends Person{  //Child / Sub class
    //TODO
}
```

- To represent is a relationship, we should use extends keyword in Java.
- In Java, parent class is called as Super class and child class is called as sub class.
- Consider Following Example

```
interface AutoCloseable{    //Perent / Super interface
    //TODO
}
```

```
interface Closeable extends AutoCloseable{  //Child / Sub interface
    //TODO
}
```

- In case of inheriatance, if super type and sub type is class the such type of inheritance is called as implementation inheritance.

```
class Person{
    //TODO
}
class Employee extends Person{  //Implementation inheritance
    //TODO
}
```

- In case of inheriatance, if super type and sub type is interface the such type of inheritance is called as interface inheritance.

```
interface AutoCloseable{
    //TODO
}
interface Closeable extends AutoCloseable{  //Interface inheritance.
    //TODO
}
```

- In OOPS(Not only in Java), there are 2 types of inheritance
  - Interface inheritance
    - Single inheritance (Java support it)
    - Multiple inheritance (Java support it)
    - Hierarchical inheritance (Java support it)
    - Multilevel inheritance (Java support it)
  - Implementation inheritance
    - Single inheritance (Java support it)
    - Multiple inheritance (Java do not support it)
    - Hierarchical inheritance (Java support it)
    - Multilevel inheritance (Java support it)

```
## Inheritance

### Types of inheritances
* Single inheritance
    ```Java
    class A {
        // ...
    }
    class B extends A {
```
```

```
        // ...
    }
    ```
* Multiple inheritance
    ```Java
    class A {
        // ...
    }
    class B {
        // ...
    }
    class C extends A, B // not allowed in Java
    {
        // ...
    }
    ```
    ```Java
    interface A {
        // ...
    }
    interface B {
        // ...
    }
    class C implements A, B // allowed in Java
    {
        // ...
    }
    ```
* Hierarchial inheritance
    ```Java
    class A {
        // ...
    }
    class B extends A {
        // ...
    }
    class C extends A {
        // ...
    }
    ```
* Multi-level inheritance
    ```Java
    class A {
        // ...
    }
    class B extends A {
        // ...
    }
    class C extends B {
        // ...
    }
    ```
* Hybrid inheritance: Any combination of above types

```
- By extending class, only sub class can reuse properties( fields ) and behaviour(
method ) of super class inside sub class.
- Super class can not reuse functionality of sub class.

- Using sub class name, we can access static field declared in super class. In
other words, static field inherit into sub class.
```java
class A{
    static int number;  //It will inherit into class B
    static {
        A.number = 100;
    }
}
class B extends A{
    static int count;     //It won't inherit into class A
    static {
        B.count = 200;
    }
}
class Program{
    public static void main(String[] args) {
        System.out.println(A.number);   //OK
        System.out.println(B.number);   //OK
    }
}
```

- Using super class name, we cannot access static field declared in sub class. In other words, static field do not inherit into super class.

```java
class A{
    static int number;  //It will inherit into class B
    static {
        A.number = 100;
    }
}
class B extends A{
    static int count;     //It won't inherit into class A
    static {
        B.count = 200;
    }
}
public static void main(String[] args) {
    System.out.println(A.count);    //NOT OK
    System.out.println(B.count);    //OK
}
```

- Access modifier do not restrict members to be inherit into sub class. In other words, all private/package level private/protected and public members inherit into sub class.
```

- All the fields(static as well as non static) of super class inherit into sub class but only non static fields get space inside instance of sub class.
- Using sub class name, we can access static method of super class. In other words, static method inherit into sub class.

```
class A{
    public static void f1( ) {
        System.out.println("super.A");
    }
}
class B extends A{

}
public class Program {
    public static void main(String[] args) {
        A.f1(); //OK
        B.f1(); //OK
    }
}
```

- Using super class name, we can not access static method of sub class. In other words, static method of sub class do not inherit into super class.

```
class A{
    public static void f1( ) {
        System.out.println("super.f1");
    }
}
class B extends A{
    public static void f2( ) {
        System.out.println("sub.f2");
    }
}
public class Program {
    public static void main(String[] args) {
        A.f2(); //Not OK
        B.f2(); //OK
    }
}
```

- In Java, constructor is not considered as a method. It is considered as a special syntax of class.
- Except constructor, all the static and non static methods of super class inherit into sub class.
- Except constructor, all the members of super class including nested type(interface/class/enum) inherit into sub class.

- If we create instance of sub class then first super class constructor gets called and then sub class constructor gets called. -From any constructor of sub class, by default super class's parameterless constructor gets called.
- Using super statement, we can call any constructor of super class from constructor of sub class.
- super statement must be the first statement inside constructor body.
- Process of acquiring/accessing properties and behavior of super class inside sub class is called as inheritance.
- During inheritance, members of sub class do not inherit into super class. Hence using super class instance we can access members of super class only.
- During inheritance, members of super class inherit into sub class. Hence using sub class instance we can access members of super class as well as sub class.
- During inheritance, members of super class, inherit into sub class. Hence we can consider sub class instance as a super class instance.
- Since sub class instance can be considered as super class instance, we can use it in place of super class instance.

```
Employee emp1 = null;   //OK
```

```
Employee emp1 = null;   //OK
emp1 = new Employee( ); //OK
```

```
Employee emp1 =  new Employee( ); //OK
```

```
Person p1 = null;   //OK
```

```
Person p1 = null;   //OK
p1 = new Person( ); //OK
```

```
Person p1 = new Person();   //OK
```

```
Person p1 = new Person();   //OK
Person p2 = p1; //OK
```

```
Employee e1 = new Employee( );
Person p1 = e1; //OK
```

```
Person p1 = new Employee(); //OK    =>  Upcasting
```

- During inheritance, members of sub class do not inherit into super class. Hence we can not consider super class instance as sub class instance.
- Since super class instance can not be considered as sub class, we can not use it in place of sub class instance.

```
Person p1 = null;   //OK
```

```
Person p1 = new Person( );  //OK
```

```
Person p1 = new Person( );  //OK
Person p2 = p1; //OK
```

```
Person p1 = new Employee( );  //OK
```

```
Employee e1 = new Employee( );
Person p1 = e1;  //OK
```

```
Employee e1 = null; //OK
```

```
Employee e1 = new Employee(); //OK
```

```
Employee e1 = new Employee(); //OK
Employee e2 = e1; //OK
```

```
Employee e2 = new Person( ); //NOT OK:  Compiler Error
```

```
Person p1 = new Person( );  //OK
Employee e2 = ( Employee )p1; //NOT OK:  ClassCastException
```

Final Conclusion

```
Person p1 = null;   //OK
Person p2 = new Person( );  //OK
Person p3 = new Employee( );    //OK
Employee e1 = null; //OK
Employee e2 = new Employee( ); //OK
Employee e3 = new Person( );    //NOT OK
```

Upcasting

- Process of converting reference of sub class into reference of super class is called as upcasting.

```
Employee emp = null;
//Person p = ( Person )emp;   //Upcasting :   OK
Person p = emp;   //Upcasting :   OK
```

```
Employee emp = new Employee( );
Person p = emp; //Upcasting :   OK
```

```
Person p = new Employee( ); //Upcasting :   OK
```

- Super class reference variable can contain reference of instance of sub class. It is also called as upcasting.
- Using upcasting we can reduce object depedendancy. Consider follwing code:

```
Person p = null;    //OK
p = new Person( );  //OK
p = new Employee( );    //OK:   Upcasting
```

Downcasting

- Process of converting reference of super class into reference of sub class is called downcasting.
- In case of upcasting, to access fields / non overriden methods of sub class then we should do downcasting.
- In case of downcasting, explicit typecasting is mandatory.

```
Person p = null;
Employee emp = ( Employee )p;   //Downcasting   :   OK
//p     =>  null
//emp   =>  null
```

```
Person p = new Employee() ; //Upcasting
Employee emp = ( Employee )p;   //Downcasting   :   OK
```

```
Person p = new Person() ; //OK
Employee emp = ( Employee )p;   //Downcasting   :   NOT OK  =>  ClassCastException
```

- If downcasting fails then JVm throws ClassCastException
- If we do upcasting then in case of following condition we should do downcasting
    - To access fields of sub class
    - To access non overriden methods of sub class.

Overriding

- Process of redefining method of super class, inside sub class using following rules is called method overriding.
- Rules of method overriding:
    - Access modifier in sub class method should be same or it should be wider.
    - Return type in sub class method should be same or it should be sub type. In other words return type in sub class method should be covariant.
    - Method name, number of parameters and type of parameters in sub class method mist be same.
    - Checked exception list in sub class method should ne same of it should be sub set.
- According to clients requirement, if implementation of super class method is logically incomplete then we should override method in sub class.

Dynamic method dispatch

- Process of calling method of sub class using reference of super class is called as dynamic method dispatch.

```
public static void main(String[] args) {
        int choice;
        while( ( choice = Program.menuList( ) ) != 0 ) {
            Shape shape = null;
```

```
            switch( choice ) {
            case 1:
                shape = new Rectangle();    //Upcasting
                break;
            case 2:
                shape = new Circle();   //Upcasting
                break;
            }
            if( shape != null ) {
                //Runtime polymorphism / Dynamic method dispatch
                shape.acceptRecord();
                shape.calculateArea();
                shape.printRecord();
            }
        }
    }
```

- According client's requirement, if implementation of super class method is logically incomplete / partially complete then we should redefine/override method inside sub class.
- Following are the rules of method overriding:
    - Access modifier in sub class method should be same or it should be wider.
    - Return type of sub class class method should be same or its should be sub type(in other words, it should be covariant).
    - Method name, number of parameters and type of parameters of sub class method must be same.
    - Checked exception list in sub class method should be same or it should be sub set.
- Override is annotation, which helps developer to override method in sub class.

```
class A{
    public void print( ){
        //TODO
    }
}
class B extends A{
    @Override   //java.lang.Override is annotation
    public void print( ){
        //TODO
    }
}
```

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).

```
class A{
    public final void f1() {
        System.out.println("A.f1");
    }
}
class B extends A{
    @Override
    public void f1() {  //Compiler error
        System.out.println("A.f1");
    }
}
```

- We can not override final method inside sub class but it gets inherited into sub class and we can call it on sub class instance.

```
class A{
    public final void f1() {
        System.out.println("A.f1");
    }
}
class B extends A{
}
public class Program {
    public static void main(String[] args) {
        B b = new B( );
        b.f1( );    //OK
    }
}
```

- Example
    - public final int ordinal( );
    - public final String name( );
    - There are 6 final methods in java.lang.Object class(getClass, wait, notify, notifyAll)
- We can not override/redefine following methods in sub class:
    - constructor
    - private method
    - static method
    - final method
- According clients requirement, if implementation of super class method is partially complete then we should not declare super class method final/abstract. In this case, we should redefine/override method inside sub class.
- We can declare overriden method final. Consider following code.

```
class A{
    public final void f1() {
        System.out.println("A.f1");
    }
    public void f2( ) {
        System.out.println("A.f2");
    }
}
class B extends A{
    public final void f2( ) {   //Overriden method
        System.out.println("B.f2");
    }
}
public class Program {
    public static void main(String[] args) {
        B b = new B( );
        b.f1( );    //OK
        b.f2();     //OK
    }
}
```

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
- we can create instance of final class but we cannot extend final class.

```
final class C extends B{
    @Override
    public final void f3() {
        System.out.println("C.f3");
    }
}
class D extends C{  //The type D cannot subclass the final class C
    //TODO
}
```

###Upcasting / downcasting

- In case of upcasting, using super class reference variable we can access
    - Fields of super class
    - methods of super class

    - overriden methods of sub class.
- In case of upcasting, using super class reference variable we can not access
    - Fields of sub class
    - non overriden methods of sub class.
- In this case to access above members we should do downcasting.
- instanceof is an operator which returns boolean value.
- using instanceof we can identify type of sub class instance.

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;
    }
}
```