Day 14

```
class A extends Exception{    }
class B extends Exception{    }
class C extends Exception{    }
public class Program {
  //public static void print( int number ) throws A, B, C {
  public static void print( int number ) throws Exception {
    if( number == 10 )
      throw new A();
    if( number == 20 )
      throw new B();
    if( number == 30 )
      throw new C();
    System.out.println("Number    :   "+number);
  }
  public static void main(String[] args) {
    try {
      Program.print(50);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

- We can not override static method inside sub class:

```
class Super{
  public static void showRecord( ) {
    System.out.println("Super.showRecord()");
  }
}
class Sub extends Super{
  @Override
  public static void showRecord( ) {    //Compiler Error
    System.out.println("Sub.showRecord()");
  }
}
```

- Non static methods are by default virtual in Java.
- In Java virtual methods are designed to call on object reference.
- Static methods are designed to call on class name. Since static methods are not designed to call on object reference it is not virtual by default. Hence we can not override static method inside sub class.

```

```

```
class Super{
  public static void showRecord( ) {
    System.out.println("Super.showRecord()");
  }
}
class Sub extends Super{
  public static void showRecord( ) {
    System.out.println("Sub.showRecord()");
  }
}
public class Program {
  public static void main(String[] args) {
    Super.showRecord(); //Super.showRecord()
    Sub.showRecord();   //Sub.showRecord()  //Due to method shadowing
/ method hiding
  }
}
```

```
  class Super{
    public void printRecord( ) {
      System.out.println("Super.printRecord()");
    }
  }
  class Sub extends Super{
    @Override
    public void printRecord( ) {    //Overrided method
      System.out.println("Sub.printRecord()");
    }
  }
  public class Program {
    public static void main(String[] args) {
      //Super s1 = new Super();
      //s1.printRecord();

      Sub s2 = new Sub();
      s2.printRecord(); //s2.printRecord()  //Due to method shadowing
/ method hiding
    }
  }
```

Rules of method overriding

- Access modifer in sub class method should be same or it should be wider.
- Return type of sub class method should covariant. It means that, return type in sub class method should be same or it should be sub type.
- Method name, number of paramters and type of parameters inside sub class method must be same.
- Checked exception list in sub class method should be same or it should be sub set.

Following methods we can not override / redefine inside sub class

- private method
- static method
- constructor
- final method

Final method final class

- According business requirement, if implementation of super class method is logically incomplete then we should override method inside sub class.
- According business requirement, if implementation of super class method is logically 100% complete then we should declare super class method final.
- final is modifier in Java.
- We can not redefine final method inside sub class. In other words, we can not override final method inside sub class.
- Final method inherit into sub class. Hence we can use it inside sub class.
- Examples of final method:
  - public final int ordinal();
  - public final String name();
  - public final Class<?> getClass();
  - public final void wait( )throws InterruptedExcption
  - public final native void wait(long timeOut)throws InterruptedExcption
  - public final void wait(long timeOut, int nanos)throws InterruptedExcption
  - public final void notify();
  - public final void notifyAll();
- We can declare overriden method final.

```
class A{
  public void f2( ) {
    System.out.println("A.f2");
  }
  public final void f3() {
    System.out.println("A.f3");
  }
}
class B extends A{
  @Override
  public final void f2() {
    System.out.println("B.f2");
  }
}
public class Program {
  public static void main(String[] args) {
    B b = new B();
    b.f2();
  }
}
```

- According business requirement, if implementation of class is logically 100% complete then we should declare such class final.
- We can not extend final class. In other words, we can not create sub class of final class.
- Examples of final class:
    - java.lang.System
    - java.lang.String
    - java.lang.StrinBuffer
    - java.lang.StringBuilder
    - All the wrapper classes
    - java.lang.Math
    - java.util.Scanner

Abstract method an abstract class

- According business requirement, if implementation of super class method is logically 100% incomplete then we should declare super class method abstract.
- abstract is modifier in java.
- We can not provide body to the abstract method.
- If we declare method abstract then we must declare class abstract.
- We can not instantiate abstract class. In other words, we can not create instance of abstract class but we can create reference of abstract class.
- Abstract class may/may not contain abstract method.
- If super class contains abstract method then sub class must override it otherwise sub class will be considered as abstract.
- Consider following code:

```
abstract class A{
  public abstract void f1( );
}
class B extends A{
  @Override
  public void f1() {
    //TODO
  }
}
```

    - Consider following code:

```
abstract class A{
  public abstract void f1( );
}
abstract class B extends A{
```

```
}
```

- Examples of abstract class:
  - java.lang.Enum
  - java.lang.Number
  - java.util.Calendar
  - java.util.Dictionary
- Examples of abstract methods:
  - public abstract int intValue();
  - public abstract float floatValue()
  - public abstract double doubleValue()
  - public abstract long longValue()
- Without declaraing method abstract, we can can declare class abstract.

```
  abstract class A{
  public abstract void f1( );
  public abstract void f2( );
  public abstract void f3( );
}
abstract class B extends A{
  @Override
  public void f1() {  }
  @Override
  public void f2() {  }
  @Override
  public void f3() {  }
}
class C extends B {
  @Override
  public void f1() {
    System.out.println("C.f1");
  }
}
class D extends B {
  @Override
  public void f2() {
    System.out.println("D.f2");
  }
}
class E extends B{
  @Override
  public void f3() {
    System.out.println("E.f3");
  }
}
public class Program {
```

```
  public static void main(String[] args) {
    A a = null;

    a = new C();
    a.f1();

    a = new D();
    a.f2();

    a = new E();
    a.f3();
  }
}
```

Sole constructor

- A constructor of super class which is designed to call from constructor of only sub class is called sole constructor.

```
abstract class A{
  private int num1;
  private int num2;
  public A( int num1, int num2) { //Sole Constructor
    this.num1 = num1;
    this.num2 = num2;
  }
  public void printRecord( ) {
    System.out.println("Num1  :   "+this.num1);
    System.out.println("Num2  :   "+this.num2);
  }
}
class B extends A{
  private int num3;
  public B( int num1, int num2, int num3 ) {
    super( num1, num2 );
    this.num3 = num3;
  }
  @Override
  public void printRecord() {
    super.printRecord();
    System.out.println("Num3  :   "+this.num3);
  }
}
public class Program {
  public static void main(String[] args) {

    B b = new B( 10, 20, 30 );
    b.printRecord();
  }
}
```

Wrapper class hierarchy

Boxing & auto-boxing

Unboxing & auto-unboxing

Generic programming