Naming Conventions

---

1. Class

Every class in java should be starts with Capital letter. If there are more than one words in a class name, then each word first letter should be capitalised.

Ex: Employee, Test, DataInput, InputStreamReader

2. method/variables/objects

Every method should starts with lower case letter. If there are more than one word in a method name, then each word first letter should be capitalised

Ex: main (), fun(), processData(), employee, age, myAge, dispEmpData()

3. Keywords/packages

Every keyword/ package name in java should be written in lower case letters.

Ex: com, class, java.lang

4. Constants

Every word in constants should be written as upper case letters.

Ex: PI, MAX_AGE, MAX_NUM

Coding conventions

---

1. Every class or method or block must be started with "{" immediately.

Ex:

```
  main(){
   }
class Test{

}
```

2. Every class name, method, variable and oject names should be defined in a meaning full way

Ex: class Person{

    int personId;

    String personName;

}

Java OOPS

---

1. class : Blue Print

    -> variables / properties / fields

    -> methods

    -> constructors

    -> blocks

    -> inner classes

properties

    -> instance

    -> static

    -> const

    -> final

Methods

    -> instance methods

    -> static methods

    -> final methods

    -> native methods

    -> abstract methods

Constructors

    -> default

    -> parametrised constructors

blocks

-> instance block

-> static block

Inner classes

-> anonymous class

-> private inner classes

-> static inner classes

Types of classes

-> concrete class

-> abstract class

-> static class

-> private class

Objects

-> state

-> behaviour

-> Class can exits with out objects. But vice versa not possible

Types of Objects

-> simple type

-> complex type

Constructor

-> its a special member block in a class

-> Invokes by JVM when the object is created.

-> Name of the class and constructor name must be same

-> constructors do not have return types. Even void not allowed

-> Constructors may have arguments

Notes

---

1. JVM provides a default constructor if you are not providing any parametarized constructor
2. When yo provide a argument constructor, then JVM will not add a default constructor automatically.
3. When you provide a argument constructor, then proving a default constructor is mandatory operation if you are trying to invoke default constructor.

Constructor Overloading

---

1. If you are trying too create an object to the sub class with out any parameters, then proving default constructor in the super and sub classes are mandatory. JVM adds them for you automatically.

2. When you provide parameterized constructor in the super class, and not proving any default constructor, then you cannot create an object to the sub class (with out parameter). In this case,

    i. Provide default constructor in the super class

    ii. Remove parametarized constructor constructor in the super class

    iii. Invoke, parameterized constructor from the default constructor o the sub class by using super keyword

3. When ever you are trying to invoke super class parametarized constructor then providing default constructor in the super is not mandatary operation.

4. Invoking the parameterized constructor of a sub class must be followed by a default constructor in the super class or removing parametarized constructor in the super class

Static methods and properties

Static data:

-> static data is sharable to every objects in all classes

Instance data:

-> instance data copy is not sharable to every object. Every objects contains a separate copy of all instance data.

Notes:

-> Static method directly can not use non static data.

-> Non static methods can access directly static and non static data

-> static methods can access only static data directly

-> we need object in a static method to access non static data

-> we need object in non static methods to access non static data of other classes

-> static data of same class can be called directly from the same class.

-> static data of other classes can be called with a class name of that static data which is having static methods

-> calling static data in a same class or different class using objects is not recommended.

System.out.println();

1. System class belongs to java.lang package
2. Out is a static object of type java.io.PrintStream class which defined in System class
3. Static data can be called with class name if that data belongs to in other class. So out can be called with a class name System.
4. By calling System.out , you will get java.io.PrintStream class object as a return type.
5. On top of java.io.PrintStream class, you can access all methods
   Of PrintStream class
6. println() method belongs to java.io.PrintStream class.
7. So we can directly invoke println method using System.out.println();

Inheritance

-> Inheriting features from super class/base class/parent class is called inheritance.

-> Creating relationships between classes

Relationships two types

1. Has 'A' -> Object relationships
2. Is 'A' -> Inheritance

Is 'A'

```
class A{
        feature 1
        feature 2

}
class T{}
class B extends A{
        class A features and
        feature 3
        feature 4
}
```

Notes

-> Only sub class can access features of super class. Vice versa is not passible

Advantages

---

1. Re-usability
2. Class relationships
3. Better data abstraction
4. Implementing polymorphism

Has 'A' relationship

If a class contains instance of an another class -> has 'A'

Notes

---

1. Creating an object to the class

        A a=new A(); creating an object

2. Instantiating to the class

```
A a=MakeObject.getObject();
A a1=MakeObject.getObject();
```

```
class MakeObject{
    public static A getObjet(){
// ----
    return new A();
  }
}
```

Types of Inheritance

Two Types

1. Single
2. Multiple

Java Supports only Single Inheritance type.

Multiple Inheritance is not allowed in Java

Inheritance Notes

---

1. Java.lang.Object class is the superclass to all the classes in Java
2. A super class object/reference can store all its sub classes/ implementation class objects.
3. Super class reference can able to access every method and properties in the super class.
4. Super class reference cannot be used to access the individual methods of its sub classes (a method which is not available in super class and only available in sub class)

Encapulation

-> Wrapping up of methods and properties in a class is called Encapsulation.

Encapsulated class

-> POJO (Plain Old Java Object)

-> Bean / Model / Entity

Rules

---

1. Class should be public
2. All properties should be defined as 'private' □
3. Every method should be defined as 'public'
4. Every method should be defined as public with getters and
   Setters
5. Getter methods do not have arguments. But they do have
   Return type
6. Setter methods do not have return type. but they do have
   Arguments. So return type always should be 'void'
7. No business logics should not be defined in POJO classes

1. Access modifiers
   -> default
   -> public
   -> protected
   -> private

2. Packages
   -> built in
   -> user defined

1. Packages

-> Groups all the classes with a specified name.

Advantages

   -> searching for classes are easy
   -> data abstract can be achieved
   -> Security can be provided
   -> classes can be organised in a proper way

1. Default access modifier

-> Accessible with in the same package only

-> no other package can not access default data of any class

Ex:

```
 package com;
  class Test{
   int age;
   String empName
    void test(){
      Use u=new Use(); // Invalid
      Employee emp=new Employee(); // Valid
    }
  }
```

```
package com;
class Employee{
          void test(){
       Test t=new Test(); // Vallid
    }
}
```

```
 package org;
 class Use{
    public void test(){
                    Test t=new Test();  // Invalid
    }
 }
```

2. Public

-> public data can be exposed to all the classes from the same

Package and to the other package classes

Ex:

```
package com;
 public class Test{
  int age;
  String empName

  void fun(){ } // default
  public void fun1(){} // public
   void test(){
    Use u=new Use(); // Invalid
    Employee emp=new Employee(); // Valid
   }
 }
```

```
package com;
class Employee{
        void test(){
     Test t=new Test(); // Vallid
   }
}
```

```
package org;
class Use{
   public void test(){
                Test t=new Test();  // Valid
      t.fun(); // Invalid
      t.fun1(); // Valid
   }
}
```

3. Protected

-> protected data is available to all the classes from the same
Package, and to all the sub class's objects of different package

Ex:

```
package com;
 protected class Test{
  int age;
  String empName

  void fun(){ } // default
  public void fun1(){} // public
  protected void fun2();
   void test(){
    Use u=new Use(); // Invalid
    Employee emp=new Employee(); // Valid
   }
 }

package com;
class Employee{
          void test(){
     Test t=new Test(); // Vallid
   }
}

 package org;
 class Use extends Test{ // Inheritance
   public void test(){
                  Test t=new Test();  // InValid
       Use u=new Use(); // Valid

            t.fun(); // Invalid
       t.fun1();// Invalid
       t.fun2(); // Invalid
```

```
        u.fun1(); // Valid
                    u.fun2();// Valid
        u.fun();// Invalid
    }
}

package org;
class Fun{
    public void test(){
                    Test t=new Test();  // Invalid
        t.fun(); // Invalid
        t.fun1(); // Valid
    }
}
```

4. Private □

-> private data is available only with in the same class.

-> Its not available to other classes with in the same package,
and to other classes with in different packages
and sub classes objects of same package and different
Packages

```
package com;
  private class Test{
  private int age;
   String empName

   void fun(){
    Test t=new Test(); // Valid
    age=98;
    } // default
   public void fun1(){} // public
```

```
  protected void fun2();
   void test(){
     Use u=new Use(); // Invalid
     Employee emp=new Employee(); // Valid
   }
 }
package com;
public class Employee{
  private String empName;
         void test(){
    empName="James";
     Test t=new Test(); // Invallid
   }
}
 package org;
 class Use extends Test{ // Invalid Inheritance
   public void test(){
                  Test t=new Test();  // InValid
       Use u=new Use(); // InValid

            t.fun(); // Invalid
       t.fun1();// Invalid
       t.fun2(); // Invalid

      u.fun1(); // InValid
                  u.fun2();// InValid
      u.fun();// Invalid
   }
 }
 package org;
 class Fun{
   public void test(){
                  Test t=new Test();  // InValid
       t.fun(); // Invalid
```

```
        t.fun1(); // InValid

       Employee emp=new Employee(); // Valid
       emp.empName="James"; // Invalid

   }
}
```

Method Overriding □

-> Defining the same method signature of a super class in the Sub class in called method overriding

Advantages

-> All sub classes can provide different implementations to the same method which Is there in the super class

-> Can apply polymorphism

Rules

---

1. A sub class can over ride super class method if super class methods are non static , non-final and non-private
2. Method over loading is not possible if the
   Super class type is final

Overriding method access visibility

---

1. If Super class method is public, then sub class must override with public access modifier only
2. If super class method is default, then sub class overridden method can be default, public and protected
3. If the super class method is protected, then sub class overridden method can be only protected and public

4. If the super class method is private. Private methods are not visible to any other classes including sub classes

Method Overloading

-> Defining the same method name with different signature

Method signature

-> name of method

-> number of arguments

-> type of arguments

-> position of arguments

Advantages?

-> readability

-> same method to perform many operations

EX: println();

Notes

-> its a static polymorphism concept. (Compile time decision)

Abstract class

-> An abstract is a method specification which contains zero or more abstract methods and concrete methods

-> An abstract class do not specify body.

-> Abstract methods must be defined only with in abstract classes

-> We cannot create object to an abstract class. But we can create a reference to an abstract class

-> The terence of an abstract class can be used to access all the methods in the abstract class and overridden in the sub classes

-> The reference of the abstract class cannot be used to access the individual methods of a sub class (a method which is available only in the sub class and not available in the super class.)

-> Every sub class of an abstract class must override every abstract method in the abstract class.

-> Any any one of abstract method is not overridden in the sub class then that sub class must be defined as 'abstract'

-> A class can be either abstract or final. It cannot be both.

Issues

---

1. Abstract classes cannot achievement 100% abstraction. We can implement only partial abstraction
2. Code is tightly coupled with abstract classes.
3. Application cannot be extended.
4. Multiple inheritance is not possible

Interfaces

-> An interface is a method specification which contains zero or more abstract methods only

-> every method of the interface by default is **public and abstract**

-> every property in the interface by default **public static final**

-> We can define a sub interface with in the interface

-> we can define a class in a interface

-> An interface which contains **no or zero abstract methods** is known as **'marker' or 'tagged' interface**

Advantages

-> extensibility

-> loosely coupled

-> 100% abstraction

-> Multiple Inheritance

Notes

---

1. Interface is not a class.
2. An interface can extends more than one interface
3. A class can implements more than one interface

4. Class can only implement interface
5. Class cannot extends interface
6. Interface only extends another interface
7. Interface can not implements another interface
8. Class cannot implements another class
9. Class can only extends to another class
10. Class can extends only one class at a time
11. Class can implements many interfaces as well cab extends one class at same time

Inner classes

-> to hide some sensitive properties / methods to other classes.

Types

---

1. Non static inner classes
2. Static inner classes
3. Private inner classes

1. non static inner classes

-> Object of this class can be created in any other class using outer class object.
-> if the inner class is default, then only the classes from same package can create object to inner class using our class obj
-> if the inner class is protected, then only the classes from same package and sub classes of outer class in other packages can create object to inner class using our class obj
-> Inner class can access every data of outer class. But outer class cannot access inner class data with out creating inner class object.

2. Static inner classes

-> Object of this class can be created in any other classes with help of outer class name. Outer class object is not needed to create object of static inner class

-> static inner class cannot access non static data of outer class.

-> static inner class can access outer class static data directly

3. Private inner classes

-> private inner class object cannot be created in any other class.

-> only outer class can create private inner class object.

-> private inner class can access data of outer class.

-> outer class cannot access data of inner class with out help of Inner class object.

Wraper classes

-> Wraping up primitives into objects and vice versa.

int x=10; -> Integer x=new Integer(10);

primitive type	Wraper class
int	Integer
short	Short
byte	Byte
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

String

-> String class is used to to String related manipulations.

-> String is a final class

-> String is a immutable class

Types of Objects

---

1. Mutable - StringBuffer, StringBuilder, and all our complex classes like Employee, Bank
2. Immutable - String , and all wraper classes