array as argument of method-constructor, access modifiers(private, public, default) for variable and method, sequence of constructor with inheritance, use of super for variable of super class
method overriding, overloading and inheritance, private method in inheritance, static varible and method in inheritance, super class reference for sub class object
instanceof, static vs. dynamic binding, dynamic method dispatch, polymorphism demo, final class and final method
defining and using a package, compile and execute from terminal, static import, access modifiers for class, constructor, method and variable.

Array as argument and return for method

```
public class ArrayAndMethod {
		int arr[];
		public ArrayAndMethod() {
		}
		 public ArrayAndMethod(int[] arr) {
			this.arr = arr;
		}
		void method1(int a[]) {
			arr = a;
		}
		void show() {
			for (int i = 0; i < arr.length; i++)
				System.out.println(arr[i]);
		 }
		public static void main(String args[]) {
			int x[] = {1, 2, 3, 4};
			ArrayAndMethod obj1 = new ArrayAndMethod(x);
			obj1.show();  // OUTPUT : 1 2 3 4
			ArrayAndMethod obj = new ArrayAndMethod();
			obj.arr = x;
			for (int i = 0; i < obj.arr.length; i++) {
				obj.arr[i]++;
			}
			obj.show(); // OUTPUT: 2 3 4 5
			 for (int i = 0; i < x.length; i++) {
				System.out.println(x[i]); // OUTPUT: 2 3 4 5
			}
		}
}
```

Access Modifiers public, private and default

```
class MyClass {
        private int a;
        int b;
        public int c;

        MyClass() {
                this.a = -1;
                this.b = -1;
                this.c = -1;
        }
        MyClass(int a, int b, int c) {
                this.a = a;
                this.b = b;
                this.c = c;
        }

        private void method1() {
                System.out.println("in private method1");
                System.out.println(a + " " + b + " " + c);
         }
         void method2() {
                System.out.println("in default method2");
                System.out.println(a + " " + b + " " + c);
        }
         public void method3() {
                System.out.println("in public method3");
                System.out.println(a + " " + b + " " + c);
        }
        public void myMethod() {
                System.out.println("in myMethod");
                this.method1();
        }
}
```

```
public class AccessModifiers {
   public static void main(String args[]) {

                MyClass obj1 = new MyClass(1, 2, 3);

                 //System.out.println(obj1.a);    //error: a has private access in MyClass
                System.out.println(obj1.b);    //OUTPUT : 2
                System.out.println(obj1.c);             //OUTPUT : 3

                 //obj1.method1();          //error: method1() has private access in MyClass
                obj1.method2();                         //OUTPUT :   in default method2
                                                                   1 2 3
                obj1.method3();                         //OUTPUT :   in default method2
                                                                   1 2 3
                obj1.myMethod();                        //OUTPUT:    in myMethod
                                                                   in private method1
                                                                   1 2 3
        }
}
```

Sequence of Constructor call with inheritance

```
class A {
                A() {
                System.out.println("Inside A's constructor.");
                }
}
class B extends A {
                B() {
                System.out.println("Inside B's constructor.");
                }
}
class C extends B {
                C() {
                System.out.println("Inside C's constructor.");
                }
}
class Demo {
                public static void main(String args[]) {
                                C c = new C();
}               }
```

OUTPUT:
Inside A's constructor
Inside B's constructor
Inside C's constructor

```
class GrandParent {
		public GrandParent() {
			System.out.println("in no arg grandparent constructor");
		}
}

class Parent //extends GrandParent{
		int x;
		public Parent(int x) {
			this.x = x;
			System.out.println("in parameterized parent constructor");
		}
		public Parent() {
			System.out.println("in no arg parent constructor");
		}
}

class Child extends Parent {
		int y;

		public Child() {
			System.out.println("in child constructor with zero parameter");
		}
		public Child(int y) {
			this.y = y;
			System.out.println("in child constructor with one parameter");
		}
		public Child(int y, int x) {
			super(x);
			this.y = y;
			System.out.println("in child constructor with one parameter");
		}

		 public void show() {
			System.out.println("x = " + x + " y = " + y + "\n");
		}
}
public class InheritanceConstructor {
	public static void main(String args[]) {
			Child c = new Child();
```

```
                        c.show();
                        Child c1 = new Child(5);
                        c1.show();
                        Child c2 = new Child(1, 2);
                        c2.show();
        }
}
```

OUTPUT:

(without GrandParent class)

```
in no arg parent constructor
in child constructor with zero parameter
x = 0 y = 0

in no arg parent constructor
in child constructor with one parameter
x = 0 y = 5

in parameterized parent constructor
in child constructor with one parameter
x = 2 y = 1
```

(with GrandParent class)

```
in no arg grandparent constructor
in no arg parent constructor
in child constructor with zero parameter
x = 0 y = 0

in no arg grandparent constructor
in no arg parent constructor
in child constructor with one parameter
x = 0 y = 5

in no arg grandparent constructor
in parameterized parent constructor
in child constructor with one parameter
x = 2 y = 1
```

Use of super to access super class variable

```
class A {
        int ai;

        public A() {
                ai = 1;
        }

         public void showA() {
                System.out.println("in showA()");
                System.out.println("ai = " + ai);
}       }

class B extends A {
        int ai;
        int bi;

         public B() {
                ai = 5;
                bi = 10;
        }
        public void showB() {
                System.out.println("in showB()");
                System.out.println("ai = " + ai + " bi = " + bi);
                System.out.println("super class ai = " + super.ai);
                this.showA();
}       }

public class SuperForVariable {
        public static void main(String args[]) {
                A objA = new A();
                objA.showA();
                B objB = new B();
                objB.showB();
                objB.showA();
}       }
```

OUTPUT:

```
in showA()
ai = 1

in showB()
ai = 5 bi = 10
super class ai = 1
in showA()
ai = 1

in showA()
ai = 1
```

Method Overriding

```
class Person {
                String name;
                 int age;
                public Person(String name, int age) {
                        this.name = name;
                        this.age = age;
                 }
                public void showDetails() {
                        System.out.println("Name = " + name + " Age = " + age);
}               }

class Student extends Person {
                int rollno;
                int marks;
                public Student(int rollno, int marks, String name, int age) {
                        super(name, age);
                        this.rollno = rollno;
                        this.marks = marks;
                }
                public void showDetails() {
                        super.showDetails();//calls superclass method
                        System.out.println("Name = " + name + " Age = " + age + " Rollno =
                        " + rollno + " Marks = " + marks);
}               }

public class MethodOverriding {
        public static void main(String args[]) {
                Student s = new Student(1, 70, "ABC", 20);
                //s.showDetails();              // of superclass  Name = ABC Age = 20
                //s.showDetail();       //of subclass Name=ABC Age=20 Rollno=1 Marks=70
                 s.showDetails();               // overidden method in sub class
}       }
```

OUTPUT:

Name = ABC Age = 20

Name = ABC Age = 20 Rollno = 1 Marks = 70

Overloading with inheritance

```
class Person1 {
        String name;
        int age;
        public Person1(String name, int age) {
                this.name = name;
                this.age = age;
        }
        public void showDetails() {
                 System.out.println("Name = " + name + " Age = " + age);
         }
}
class Student1 extends Person1 {
        int rollno;
         int marks;
        public Student1(int rollno, int marks, String name, int age) {
                super(name, age);
                this.rollno = rollno;
                this.marks = marks;
        }
        @Override
        public void showDetails() {
                System.out.println("hi");
        }

        public void showDetails(int i) {
                this.showDetails();     //calls superclass method if not overridden
                                        //else call same class method
                super.showDetails();//calls superclass method
                System.out.println("i = " + i + " Name = " + name + " Age = " + age
                + " Rollno = " + rollno + " Marks = " + marks);
        }
}
public class MethodOverloadingInheritance {
        public static void main(String args[]) {
                Student1 s = new Student1(1, 70, "ABC", 20);
                s.showDetails();
                 s.showDetails(0);
        }
}
```

OUTPUT:

```
hi
hi
Name = ABC Age = 20
i = 0 Name = ABC Age = 20
Rollno = 1 Marks = 70
```

Rules for method overriding:

- In java, a method can only be written in Subclass, not in same class.
- The argument list should be exactly the same as that of the overridden method.
- The return type should be the same or a subtype of the return type declared in the original overridden method in the super class.
- The access level cannot be more restrictive than the overridden method's access level.
  For example: if the super class method is declared public then the over-ridding method in the sub class cannot be either private or protected.
- Instance methods can be overridden only if they are inherited by the subclass.
- A method declared final cannot be overridden.
- A method declared static cannot be overridden but can be re-declared.
- If a method cannot be inherited then it cannot be overridden.
- A subclass within the same package as the instance's superclass can override any superclass method that is not declared private or final.
- A subclass in a different package can only override the non-final methods declared public or protected.
- An overriding method can throw any uncheck exceptions, regardless of whether the overridden method throws exceptions or not.
  However the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. The overriding method can throw narrower or fewer exceptions than the overridden method.
- Constructors cannot be overridden.

Static vs. Dynamic binding (Connecting a method call to the method body is known as binding.) There are two types of binding

1. **Static Binding (also known as Early Binding):**
   - When type of the object is determined at compile time, it is known as static binding.
   - Private, static and final methods are bound at compile time and they cannot be overridden.
   - Overloaded methods are bound at compile time.
2. **Dynamic Binding (also known as Late Binding).**
   - When type of the object is determined at run time, it is known as dynamic binding.
   - Overridden methods are bound at runtime.

The **java instanceof operator** is used to test whether the object is an instance of the specified type (class or subclass or interface).

- The instanceof in java is also known as type *comparison operator* because it compares the instance with type.
- It returns either true or false.
- If we apply the instanceof operator with any variable that has null value, it returns false.

Polymorphism Demo

```
class A {
        int i = 5;
        void show()    {
                System.out.println("show method of A");
        }
        void showA() {
                System.out.println("showA method of A");
        }
}

class B extends A {
        int i = 10;
        @Override
        void show()    {
                System.out.println("show method of B");
        }
        void showB() {
                System.out.println("showB method of B");
        }
}

public class PolymorphismDemo {
        public static void main(String args[]) {
                A obj1 = new A();
                System.out.println(obj1.i);     //5
                obj1.show();                    //show method of A
                obj1.showA();                   //showA method of A
                //obj1.showB();                 //cannot find symbol
                //((B) obj1).showB();           //Exception in thread "main"
                                                 java.lang.ClassCastException: A cannot be cast to
                                                B
                B obj2 = new B();
                System.out.println(obj2.i);     //10
                obj2.show();                    //show method of B
                obj2.showA();                   //showA method of A
                obj2.showB();                   //showB method of B
```

```
            A obj3 = new B();
            System.out.println(obj3.i);     //5 why?? polymorphism is only for methods
                                            not for data members
            obj3.show();                    //show method of B
            obj3.showA();           //showA method of A
            //obj3.showB();                 //cannot find symbol
            ((B) obj3).showB();             //showB method of B
        }
}
```

final class cannot be inherited. Many classes in java library are final.

```
final class MyFinalClass {
        void method() {
        }
}
```

class DeriveFromMyFinalClass extends MyFinalClass {

- **cannot inherit from MyFinalClass**
- **so this class cannot be made**

}

Use of final with method to prevent overriding

```
class MyClass1 {
        final void method1() {
                System.out.println("in final method1 of MyClass1");
        }
        void method2() {
                System.out.println("in method2 of MyClass1");
        }
}

class MySubClass1 extends MyClass1 {
        public void show() {
                this.method1();
                this.method2();
                super.method1();
                super.method2();
        }
```

```
	@Override
	void method2() {
		System.out.println("in method2 of MySubClass1");
	}

	/*void method1() {}   // does allow to write as method1() of MyClass1 is final
}

public class UsingFinal {
	public static void main(String args[]) {
		MySubClass1 obj = new MySubClass1();
		obj.show();
	}
}
```

OUTPUT:
```
in final method1 of MyClass1
in method2 of MySubClass1
in final method1 of MyClass1
in method2 of MyClass1
```

Packages

Class name must be unique to avoid name collisions. Java provides a mechanism called package for partitioning the class name space. The package is both a naming and a visibility control mechanism.

You can define classes inside a package that are not accessible by code outside that package. You can also define class members that are accessible only to other members of the same package.

So related classes can have knowledge of each other and for rest of the world this knowledge is hidden.

Defining a Package

To create a package, write package statement as first statement of the java source file. Any classes declared within that file will belong to the specified package. The package statement defines a namespace in which classes are stored.

If the package statement is omitted, the class names are put into the default package, which has no name.

package mypackage; this statement will create a package of the name mypackage.

Java uses file system directories to store packages. For example, the .class files for any classes you declare to be part of **mypackage** must be stored in a directory called **mypackage**. It is case sensitive and directory name must exactly match the package name.

You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a dot. A package hierarchy must be reflected in the file system of your Java development system.

For example, a package declared as **package java.awt.image;** needs to be stored in **java\awt\image** in a Windows environment.

Note: You cannot rename a package without renaming the directory in which the classes are stored.

Package names are written in all lower case to avoid conflict with the names of classes or interfaces.

How does Java runtime system know where to look for packages??

1. By default, java runtime system uses the **current working directory** as its starting point. i.e. if your package is in a sub directory of the current directory, it will be found.
2. Specify a directory path/s by setting the **CLASSPATH environment variable**.
3. Use the **–classpath option** with java and javac to specify the path to your class.

For 2nd and 3rd options,
If path to MyPack package is C:\MyPrograms\Java\MyPack, then classpath will be CL\MyPrograms\Java.
In linux,
export CLASSPATH=/home/user/classdir:.:/home/user/archives/archive.jar
In Windows,
set CLASSPATH=c:\classdir;.;c:\archives\archive.jar

Example:

```
package MyPack;

class MyClass{
        public static void main(String args[]) {
                System.out.println("MyClass class is in MyPack package");
        }
}
```

This class is saved at location /home/user1/MyPack.
To compile and run,

```
/home/user1> javac MyPack/MyClass.java
/home/user1> java MyPack.MyClass
```

Or

```
/home> java –cp /home/user1 MyPack.MyClass
```

A class can use all classes from its own package and all public classes from other packages.
You can access the public classes in another package in two ways.

1. The first is simply to add the full package name in front of every class name.
   For example:
   java.util.Scanner sc = new java.util.Scanner();

2. The simpler, and more common, approach is to use the import statement. The point of the import statement is to give you a shorthand to refer to the classes in the package. Once you use import, you no longer have to give the classes their full names. You can import a specific class or the whole package.

   import java.util.*;
   import java.util.Date;

You place import statements at the top of your source files (but below any package statements).

If the import statements have two classes with same name from different packages, then full package name must be used each time the class name is used.

```
import mypack.first.HelloWorld;
import mypackage.pack1.HelloWorld;

HelloWorld obj = new HelloWorld();      Which HelloWorld will be considered??
```

static import

A form of the import statement permits the importing of static methods and fields, not just classes.
For example, if you add the directive
import static java.lang.System.*;
to the top of your source file, then you can use the static methods and fields of the System class without the class name prefix:
out.println("Goodbye, World!"); // i.e., System.out
exit(0); // i.e., System.exit

You can also import a specific method or field:
import static java.lang.System.out;

Access Modifiers for access protection

	class	Constructor / Variable / Method
private	Not allowed	Allowed but not used as it allows object creation in same class only. Object creation is not allowed in different class even in same package.
default	• Can be accessed in same package. • This class can be written in any .java file	• Can create object in same package • Does not allow inheritance into subclass outside package.
protected	Not allowed	• Can create object in same package • Allows inheritance into subclass outside package • Does not allow object creation outside package
public	• Can be accessed anywhere • This class must be written in .java file with same name as class name	Allows everything

MyClass.java

```
package pack1;

public class MyClass {

    private int pri;
    int def;
    protected int pro;
    public int pub;

    private void pri_method() {
        /*
         * this method can be accessed only in same class. for simplification
         * private methods can be written which are helper methods
         * they cannot be called out the class using object.
         */
    }

    void default_method() {
    }

    protected void pro_method() {
    }

    public void pub_method() {
    }

    public static void main(String args[]) {
        MyClass obj = new MyClass();
        obj.pri = 1;
        obj.def = 1;
        obj.pro = 1;
        obj.pub = 1;

        obj.pri_method();
        obj.default_method();
        obj.pro_method();
        obj.pub_method();
    }
}
```

SamePack.java

```
package pack1;

public class SamePack {

   public static void main(String args[])
   {
      MyClass obj = new MyClass();
      //obj.pri=1; only in same class
      obj.def=1;
      obj.pro=1;
      obj.pub=1;
      //obj.pri_method(); only in same class
      obj.default_method();
      obj.pro_method();
      obj.pub_method();
   }
}
```

SubClassSamePack.java

```
package pack1;

public class SubClassSamePack extends MyClass {

   public static void main(String args[])
   {
      SubClassSamePack obj = new SubClassSamePack();
      //obj.pri=1; //only in same class
      obj.def=1;
      obj.pro=1;
      obj.pub=1;
      //obj.pri_method(); only in same class
      obj.default_method();
      obj.pro_method();
      obj.pub_method();
   }
}
```

SubClassDiffPack.java

```
package pack2;

public class SubClassDiffPack extends pack1.MyClass{

   public static void main(String args[])
   {
      SubClassDiffPack obj= new SubClassDiffPack();

      //obj.pri=1; only in same class
      //obj.def=1; only in same package
      obj.pro=1;
      obj.pub=1;
      //obj.pri_method(); only in same class
      //obj.default_method(); only in same package
      obj.pro_method();
      obj.pub_method();
   }
}
```

DiffPack.java

```
package pack2;

public class DiffPack {

   public static void main(String args[])
   {
      pack1.MyClass obj = new pack1.MyClass();
      //obj.pri=1; only in same class
      //obj.def=1; only in same package
      //obj.pro=1; only in same package or sub class
      obj.pub=1;

      //obj.pri_method(); only in same class
      //obj.default_method(); only in same package
      //obj.pro_method();  only in same package or sub class
      obj.pub_method();
   }
}
```