Java Programming

Follow:

Blog: www.rkkeynotes.blogspot.com

YouTube: RK Keynotes

Instagram: RK KEYNOTES (@rk_keynotes)

LinkedIn: https://www.linkedin.com/in/ravikumarrn/

Facebook: https://www.facebook.com/Ravikumar.R.natarajan

Regards,

Ravikumar R N

RK KEYNOTES

1.Inheritance

It is defined as the process where derived class can borrow the properties of base class.
Inheritance can be achieved by using the "**extends**" keyword

Advantages:

i. Code reusability
ii. Extensibility
iii. Data hiding
iv. Overriding

Super class – Base class – Parent class

Sub class – Derived class – Child class (These classes inherit properties from Super or Base or Parent class)

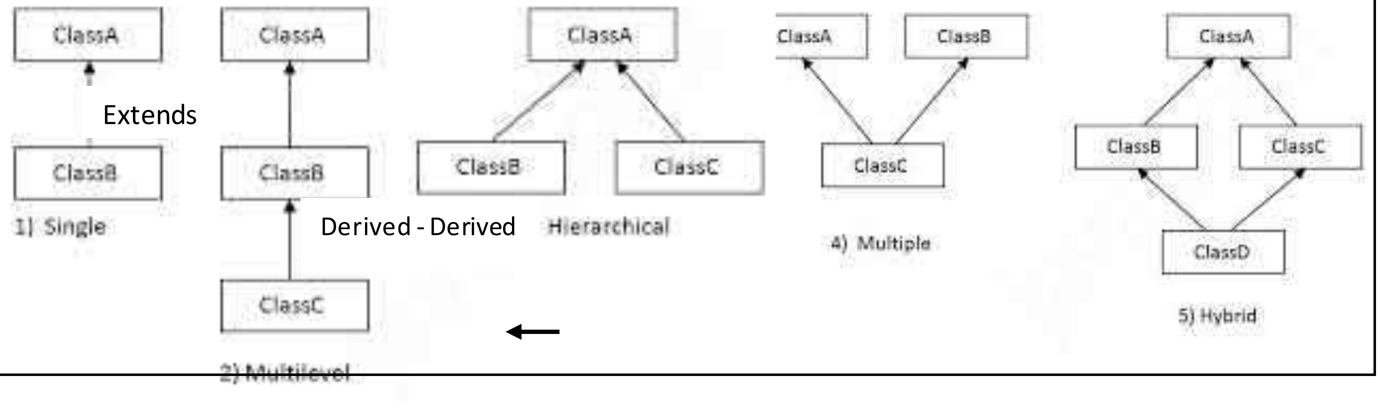
Example:

```
Class A                 //Base class //Super class // Parent class
{
}
Class B extends A       //Derived class //Sub class //Child class
{
..//Use of class A properties
}
```

Types of inheritance

1. **Single inheritance** - It has one parent per derived class. | Whenever a class inherits another class, it is called single inheritance.
2. **Multilevel inheritance** – when a derived class is derived from base class which itself is derived again is called multilevel inheritance. | Multilevel inheritance is a part of single inheritance where more than two classes are in the sequence.
3. **Hierarchical inheritance** - If a class has more than one derived classes This is known as hierarchal inheritance.
4. **Multiple inheritance** - If a class inherits the data member and member function from more than one base class, then this type of inheritance is called multiple inheritance.
5. **Hybrid inheritance** – when two or more types of inheritances(Single and multiple inheritance) are combined together then it is called hybrid inheritance.

RK KEYNOTES



1. **Single inheritance – Example**

```
class a              //base class
{
    void disp()      //method 1
    {
            System.out.println("Hello");
    }
}
class b extends a    //derived class
{
    void disp1()     //method 2
    {
    System.out.println(" Good morning");
    }
}
class singleinh      //main class
{
    public static void main(String args[])         //main function
    {
    b B1 = new b();         // reference variable A1, Creating object to allocate memory space
    B1.disp1();             //calling method
    B1.disp();
    }
}
```

2. **Multilevel inheritance – Example**

```
    class a
    {
            void disp()
            {
                    System.out.println("Helllo");
            }
    }
    class b extends a
    {
            void disp1()
            {
            System.out.println("Good morning");
            }
    }
    class c extends b
    {
            void disp2()
            {
            System.out.println("Welcome to BEC");
            }
```

RK KEYNOTES

```
}
class multilevelinh
{
        public static void main(String args[])
        {
        c C1 = new c();
        C1.disp2();
        C1.disp1();
        C1.disp();
        }
}
```

3. Hierarchical inheritance – Example

```
class a
{
        void disp()
        {
                System.out.println("Hello");
        }
}
class b extends a
{
        void disp1()
        {
        System.out.println("Good morning");
        }
}
class c extends a
{
        void disp2()
        {
        System.out.println("Welcome to BEC");
        }
}
class hierainh
{
        public static void main(String args[])
        {
        c C1 = new c();
        C1.disp2();
        C1.disp();
        //C1.disp1(); //Compile time error
        }
}
```

Why Interface should be used in multiple inheritance? Consider the example

```
class A
{
        void msg()
        {
```

RK KEYNOTES

```
            System.out.println("Hello");
            }  }
class B
{
            void msg()
            {
            System.out.println("Welcome");
            }  }
class C extends A,B //suppose if it were
{
   public static void main(String args[])
   {
            C obj=new C();
            obj.msg();//Now which msg() method would be invoked?
    }  }
```

Compile time error occurs

4. **Multiple inheritance using interface - Example**

```
interface vehicleone
{
            int  speed=90;
            public void distance();
}
interface vehicletwo extends vehicleone
{
            int distance=200;
            public void speed();
}
class Vehicle  implements vvehicletwo //two interfaces
{
            public void distance()
            {
                        int  distance=speed*200;
                        System.out.println("distance travelled is "+distance);
            }
            public void speed()
            {
                        int speed=distance/90;
                        System.out.println("Speed is "+speed);
            }
}
class MultipleInterface{
            public static void main(String args[]){
                        Vehicle obj = new Vehicle();
                        obj.distance();
                        obj.speed();}   }
```

RK KEYNOTES

5. Hybrid Inheritance using interface – Example

```
class A                                 //class
{
    int a=10;                           //variable with value assigned
}

interface B                             //Interface
{
int b=20;
}

class C extends A implements B          //Multiple inh.
{
int c;
int mul()                               //method1
{
    c=a*b;
    return c;
}
}

class D extends C                       //Single inh.
{
void sum()                              //method2
    {
    System.out.println("Adding all 3 variables");
    int d=a+b+mul();                    //Calc
    System.out.println(d);
    }
}
class hybridinh                         //main class
{
public static void main(String[] args)  //main fun.
    {
    C obj1 = new C();                   //Obj. creation
    D obj2 = new D();
    System.out.println("Multiplying two variables");
    System.out.println(obj1.mul());     //Print stmt
    obj2.sum();                         //Calling method
    }
}
```

RK KEYNOTES

2.Constructors in sub classes

- A constructor invokes its superclass constructor explicitly (means defined by programmer).
- Calling of constructor occurs from top to down.

Example:

```
class AA
{
public AA()
          {
          System.out.println("1. Statement in class AA");
          }
}
class BB extends AA
{
public BB()
          {

          System.out.println("2. Statement in class BB");
          }
}
class CC extends BB
{
public CC()
          {

          System.out.println("3. Statement in class CC");
          }
}
class chain_const
{
public static void main(String args[])
{
          CC obj = new CC();
}
}
```

Output:

```
1. Statement in class AA
2. Statement in class BB
3. Statement in class CC
```

RK KEYNOTES

2.1.Constructor chaining

Constructor chaining is the process of calling one constructor from another constructor with respect to current object.
Two ways:

- **Within same class**: It can be done using **this()** keyword for constructors in same class
- **From base class:** by using **super()** keyword to call constructor from the base class.

```
class Temp
{

    Temp()
    {
        // calls constructor 2
        this(5);
        System.out.println("The Default constructor");
    }

        Temp(int x)
    {
        // calls constructor 3
        this(5, 15);
        System.out.println(x);
    }
     Temp(int x, int y)
    {
        System.out.println(x * y);
    }
     public static void main(String args[])
    {
        // invokes default constructor first
        new Temp();
    }
}
```

Output:

```
75
5
The Default constructor
```

RK KEYNOTES

3. Super classes – Using super keyword

The subclass constructor uses the keyword super to invoke the constructor method of super class (Parent class).
The **super** keyword in java is a reference variable that is used to refer parent class objects.
Syntax: super(Parameter-list);

Conditions where super key word used:
1. Only used in subclass constructor and methods.
2. Call to super must first statement in subclass constructor.
3. Parameters must be in same order.

Example:
3.1. Use of super with variables:

```
class Vehicle
{
   int maxSpeed = 120;
}

class Car extends Vehicle
{
   int maxSpeed = 180;

   void display()
   {
      System.out.println("Maximum Speed: " + super.maxSpeed);
   }
}

class Test
{
   public static void main(String[] args)
   {
      Car small = new Car();
      small.display();
   }
}
```

Ouput:

```
Maximum Speed: 120
```

RK KEYNOTES

3.2. Use of super with methods:

```
class Person
{
   void message()
   {
      System.out.println("This is person class");
   }
}
 class Student extends Person
{
   void message()
   {
      System.out.println("This is student class");
   }

   void display()
   {
       super.message();
       message();

   }
}
 class Test
{
   public static void main(String args[])
   {
      Student s = new Student();
       s.display();
   }}
```

Output:

```
This is person class
This is student class
```

3.3. Use of super with constructors: super keyword can also be used to access the immediate parent class constructor. One more important thing is that, ‘’super’ can call both parametric as well as non parametric constructors depending upon the situation.

```
class Person
{
   Person()
   {
      System.out.println("Person class Constructor");
   }

   Person(int i)
   {
      System.out.println("Person class Const. with param");
   }
```

RK KEYNOTES

```
}
 class Student extends Person
{
    Student()
    {

         System.out.println("Student class Constructor");
    }
    Student(int i)
    {
         super(i);
         System.out.println("Student class Const. with param");
    }
}
 class Testsuper
{
    public static void main(String[] args)
    {
        Student s = new Student(5);
    }
}
```

Output:

```
Person class Const. with param
Student class Const. with param
```

RK KEYNOTES

4. Method overloading and Method overriding

4.1.Method overriding: - (Subclass inherits method of super class) | Occurs in run time.

- Declaring a method in **sub class** which is already present in **parent class** is known as method overriding. | performed between two classes.
- Overriding is done so that a **child class can give its own implementation** to a method which is already provided by the parent class.
- In this case the method in parent <u>class is called overridden method</u> and the <u>method in child class is called overriding method</u>.

Example : 1 - for method overriding:

```
class Human
{
        public void eat()     //Overridden method
        {
        System.out.println("Human is eating");
        }
}
class Boy extends Human
{
        public void eat()     //Overriding method
        {
        System.out.println("Boy is eating");
        }
 public static void main( String args[])
  {
     Boy obj = new Boy();
    //This will call the child class version of eat()
    obj.eat();
  }}
```

Output:

```
Boy is eating
```

Example 2: for method overriding using super

```
class AA
{
        int a=0;
void fun(int i)
        {
        this.a=i;
        }}
```

Ex3:
```
class Animal {
   public void move() {
      System.out.println("Animals can move");
   }
}

class Dog extends Animal {
   public void move() {
      System.out.println("Dogs can walk and run");
   }
   public void bark() {
      System.out.println("Dogs can bark");
   }
}
public class TestDog {
   public static void main(String args[]) {
      Animal a = new Animal();
      Animal b = new Dog();
      a.move();   // runs the method in Animal class
      b.move();   // runs the method in Dog class
      b.bark(); // Error – cz method doesn't exist in
parent
   }}
```

Ex4:
```
class parent
{
        public void property()      //Overridden method
        {
            System.out.println("Land+Property+Cash");
        }

        public void marriage()
        {
           System.out.println("Muniamma");
        }
}
class son extends parent
{
        public void marriage()
```

RK KEYNOTES

```
class BB extends AA
{
            int b=20;
void fun(int i)
            {
            super.fun(i+5);
            System.out.println("Value of a:"+a);
            System.out.println("Value of b:"+b);
            }}
class override_demo
{           public static void main(String args[])
            {           BB b1 = new BB();
                        b1.fun(10);       } }
```

Rules:

1. Private method cannot be overridden. (so don't use private access specifier)
2. Static method can be inherited but cannot be overridden
3. Method overriding occurs only when name of two methods are same.
4. **Arguments** must be **same** and in same order from both child and parent class.

4.2.Method Overloading: (Occurs in compile time)

- It means many **methods** can **have same name but** can pass **different** number of **parameters**
- **Method Overloading** is a feature that allows a class to have more than one **method** having the same name, if their argument lists are different.
- It is similar to constructor **overloading.**

Example for method overloading:

```
class MethodOverloading
{
   private static void display(int a)
            {
      System.out.println("Arguments: " + a);
            }
   private static void display(int a, int b)
            {
      System.out.println("Arguments: " + a + " and " + b);
            }
   public static void main(String[] args)
            {
      display(1);
      display(1, 4);}}
```

Example 02: method overloading

```
      public class Sum
      {
            public int sum(int x, int y)
            {
                        return (x + y);
            }
            public int sum(int x, int y, int z)
            {
                        return (x + y + z);
            }
```

RK KEYNOTES

```
    public static void main(String args[])
        {
                Sum s = new Sum();
                System.out.println(s.sum(10, 20));
                System.out.println(s.sum(10, 20, 30));
        }
    }
```
Rules:

1. Performed with in class
2. Functions may have different return types.

5.Abstract classes and methods: (cannot be instantiated)

Hiding the internal implementation of the feature and only showing the functionality to the users. i.e. what it works (showing), how it works (hiding). Both abstract class and interface are used for abstraction.

Example for abstract class and method:

```
abstract class Animal                    //abstract parent class
{
  public abstract void sound();            //abstract method
// Public abstract void disp();          //if this method is not overridden then class error occurs
}
public class Dog extends Animal
{
          public void sound()
          {
                    System.out.println("Woof");
          }
          public static void main(String args[])
          {
                    Animal obj = new Dog();
                    obj.sound();
          }
}
```
Rules:

1. Abstract method must present in abstract class only.
2. Cannot be instantiated i.e not allowed to create object.
3. Method must be overridden. | It can have constructors & final and static methods

Difference between Abstract class and Interface

S.No	Abstract class (Cannot be instantiated)	Interface (Cannot be instantiated)
1.	**Partial** abstraction	**100**% data abstraction
2.	Programmer knows 50% of implementation	Programmer doesn't aware of implementation.
3.	A class can **inherit** only **one** abstract class.	A class can **implement more than one** interface

RK KEYNOTES

4.	Methods may or may not have implementation.	Methods have no implementation.
5.	Efficient performance	Slow performance
6.	**Extends** keyword used	**Implements** keyword
7.	Members of abstract class can have **public**, **private**, **protected**.	Members of interface are **public** by default

6.Final methods and classes

A **class** declared as **final** cannot be extended while a method declared as **final** cannot be overridden in its subclasses.

Final keyword can be applied in 3 places,

1. For declaring variables
2. For declaring methods
3. For declaring the class

Example: final keyword for declaring method

```
Class test
{
final void fun()
        {
        System.out.println("function using final");
        }
}
Class test1 extends test
{
final void fun()
        {
        System.out.println("another function using final");
        }
}
```

Output: Error cannot override fun()

Example: Final keyword with variable

```
class Demo{

  final int MAX_VALUE=99;
  void myMethod(){
    MAX_VALUE=101;
  }
  public static void main(String args[]){
    Demo obj=new  Demo();
    obj.myMethod();
  }
}
```

Output: Compile time error

Example: final class to stop inheritance

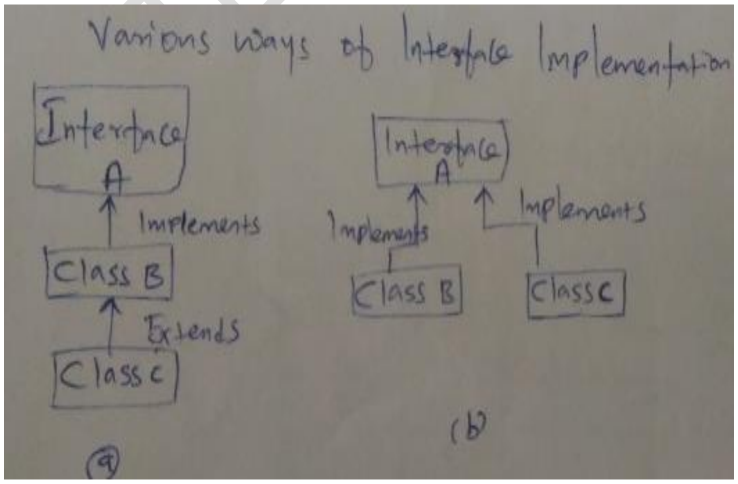
```
Final class test
{
void fun()
        {
        System.out.println("function using final");
        }
}
Class test1 extends test                    //error  occurs here
{
```

RK KEYNOTES

```
final void fun()
            {
            System.out.println("another function using final");
            }
}
```

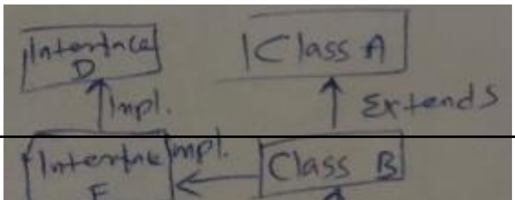
7.Implementing and Extending interface

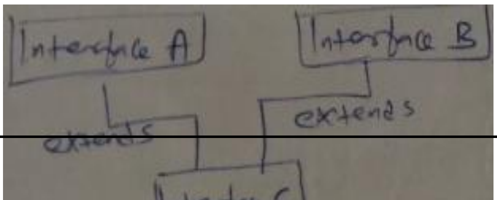
Like classes, interfaces can also be extended. i.e an interface can be sub-interfaced from other interfaces. "Extends " keyword is used.



Example for extending interface:

```
     interface Interface1
     {
         public void f1();
     }

     interface Interface2 extends Interface1
     {
         public void f2();
     }

     class x implements  Interface2
     {
         public void f1()
        {          System.out.println("Contents of Method f1() in Interface1");     }

         public void f2()
        {         System.out.println("Contents of Method f2() in Interface2");    }

          public void f3()
        {        System.out.println("Contents of Method f3() of Class X");    }}

        class ExtendingInterface
     {
          public static void main(String[] args)
           {
            Interface2 v2; //Reference variable of Interface2
            v2 = new x();  //assign object of class x
```





```
          v2.f1();

          v2.f2();

          x xl=new x();

          xl.f3();
        }
    }
```
Output:

8.D/B Class or Concrete class and interface

S.No	Class or Concrete class	Interface
1.	Keyword class used	
2.	It contains data members and methods. Methods are defined in class.	
3.	Can be instantiated	
4.	Class can use public, private, or protected	

Example: Interface and Abstract Class

```
interface A{
void a();
void b();
void c();
void d();
}

abstract class B implements A{
public void c(){System.out.println("I am c");}
}

class M extends B{
public void a(){System.out.println("I am a");}
public void b(){System.out.println("I am b");}
public void d(){System.out.println("I am d");}
}

class Test5{
public static void main(String args[]){
```

RK KEYNOTES

```
    A a=new M();
    a.a();
    a.b();
    a.c();
    a.d();
    }}
Output:I am a
       I am b
       I am c
       I am d
```

9.The Object class

The **Object class** defines the basic state and behavior that all **objects** must have, such as the ability to compare oneself to another **object**, to convert to a string, to wait on a condition variable, to notify other **objects** that a condition variable has changed, and to return the **object's class**. Object is a superclass of all the other classes by default.

Methods of Object class

The Object class provides many methods. They are as follows:

Method	**Description**
public final Class getClass()	returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
public int **hashCode()**	returns the hashcode number for this object.
public boolean **equals**(Object obj)	compares the given object to this object.
protected **Object clone()** throws CloneNotSupportedException	creates and returns the exact copy (clone) of this object.
public String **toString**()	returns the string representation of this object.
public final void notifyAll()	wakes up all the threads, waiting on this object's monitor.
public final void **wait()**throws InterruptedException	
protected void **finalize()**throws Throwable	is invoked by the garbage collector before object is being garbage collected.

RK KEYNOTES

Example:

class A **extends Object**

{ }

Class B extends A

```
{  }
class objClassDemo
{
public static void main(String args[])
{
        A obj = new A();
        System.out.println("obj:" +obj); //or +obj.toString()
}}
Output: A@3e25a5  //ClassName@HexaDecimalCode
```

10.Object cloning (duplicating or copying the object)

Object cloning is a way to create exact copy of an object. The **clone()** method of object class is used to clone an object. 2 classes (**java.lang.Object** and **ClassToClone**) and 1 interface (**java.lang.Cloneable**) used in cloning. If not cloneNotSupportedException occurs.

RK KEYNOTES

Advantages:

1. Lengthy and repetitive codes minimized
2. Easy to clone objects from old projects
3. Clone() is fastest way to copy arrays

Disadvantages:

1. To use **clone()** have to change lot of syntax
2. Implement cloneable interface when it doesn't have method. Have to inform JVM about object cloning.
3. Object.clone() only supports shallow copying. Deep cloning can be done by overriding.

Example:

```
class Student implements Cloneable // impl.Cloneable interface to indicate Object.clone()
{                                          // method  is legal to use and copy fields
   int rollno;
   String name;

Student(int rollno,String name)
        {
          this.rollno=rollno;
         this.name=name;
         }
     public Object clone()throws CloneNotSupportedException
{
    return super.clone();
  }

 public static void main(String args[])
{      try
          {
          Student s1=new Student(101,"amit");

          Student s2=(Student)s1.clone();

           System.out.println(s1.rollno+" "+s1.name);
           System.out.println(s2.rollno+" "+s2.name);

          }catch(CloneNotSupportedException c){}
        }
   }
```
Output:
```
101 amit
101 amit
```

Shallow copy – Two objects but one memory
Deep copy – Two objects Two different memory

11.Inner class

- **Java inner class** or nested class is a class which is declared inside the class or interface.

- We use inner classes to logically group classes and interfaces in one place so that it can be more readable and maintainable.

-Additionally, it can access all the members of outer class including private data members and methods.

Advantage of java inner classes

1) Nested classes **can access all the members (data members and methods) of outer class** including private.
2) T**o develop more readable and maintainable code**
3) **Code Optimization**

Types of Nested classes

There are two types of nested classes non-static and static nested classes.

1. Non-static nested class (inner class)
   - Member inner class
   - Anonymous inner class
   - Local inner class
2. Static nested class(Static inner class)

Type	**Description**
Member Inner Class	A class created within class and outside method.

RK KEYNOTES

Anonymous Inner Class	Situations: Created when class has short body | only one instance class | class is under immediately after defining it. **Anonymous classes** are inner classes with no name.
Local Inner Class	A class created within method.
Static Nested Class	A static class created within class.
Nested Interface	An interface created within class or interface.

Static inner class

```
class TestOuter1{
    static int data=30;
    static class Inner{
    void msg(){System.out.println("data is "+data);}
  }
  public static void main(String args[]){
    TestOuter1.Inner obj=new TestOuter1.Inner();
    obj.msg();
  }  }
Output:
data is 30
```

Anonymous classes

```
class Myclass implements Runnable
{
Public void run()
{
S.o.p("Hi");
}
Class DemoClass
{
Psvm(s[] a)
{
Myclass m1 = new Myclass();
Thread th = new Thread(m1);
}}}
```

15.Collection Framework - Array List (Grow able array)

The collection framework provides a well designed set of interface and classes for storing and manipulating group of data as a single unit. (Collection classes are AbstractList, LinkedList, ArrayList, HashMap, TreeMap, TreeSet etc.,)

The **ArrayList** class implements the List interface. It is used to implement dynamic array. The size of array extended automatically. When objects removed the size will be reduced. The **syntax**,

- **ArrayList**() – Creates empty array
- **ArrayList**(Collection collection) – creates list and add elements
- **ArrayList**(int c) – creates list with specified capacity c.

Methods of AL

- Void add() – insert element at specified position
- Void clear() – remove all elements from list
- Object[ ] to array() -
- Object clone()
- Void trimToSize()

Advantages:

1. AL can contain **duplicate** elements
2. AL maintains **insertion** order
3. It is non-synchronized
4. Allows **random** access

Disadvantages:

1. Manipulation (Remove, Replace) is **slow** because lot of shifting needed.

RK KEANOT

Example of AL

```
Import java.util.ArrayList
class TestCollection
{
public static void main(String args[])
{
ArrayList <String> list = new ArrayList<String> ();
list.add("Hi");
list.add("Hello");
Iterator itr = list.iterator();
while(itr.hasNext())
{
System.out.println(itr.next());
}}}
```

Iterator:
It is a **Java** Cursor used to iterate a collection of objects. It is used to traverse a collection object elements one by one. Compare to Enumeration(Used in JDK 1.0) interface, **Iterator** method names are simple and easy to use.

16.Strings

String is a collection of characters. | Sequence of Unicode characters treated as single unit. In java string defines the object.

String constructors

- String() - represents empty char sequence
- String(byte[] byte) - constructs new string by decoding array
- String(byte[] bytes, int offset) – decoding specified subarray
- String(String strObj) - constructs string obj to same sequence.

String Literal

- Represented as sequence of characters enclosed in double quotes.
- Ex: String s = "Hello"

Operations on String

Method	**Description**
s.charAt(position)	Returns char present position
s.compareTo(s2)	s<s2, s>s2, s=s1
s.concat(s2)	Join s and s1 strings
s.equals(s2)	If both equal return true
s.indexof('c')	Returns first occurrence of char
s.length()	Gives length of char
String.Valueof(var)	Converts value of variable to string
s.equalsIgnoreCase(s2)	Ignore and equals

RK KEYNOTES

Creating Strings

Two ways, i. Using the new Ex: String s = new String("Hello");

ii. Using string literals Ex: String s = "Hello"

Immutable string (Cannot change obj)

- String class represents immutable string that means once one has created a string object it cannot be changed.
- If want to modify need to use StringBuffer class.

Ex: immutability concept

```
class a
{
psvm(s[] a)
{
String s = "Hi";
s.concat("Hello"); //appends string at end
s.o.p(s); //prints only Hi because strings are immutable.
}}
```

String concatenation

1. **By using +**
```
class x
{
public static void main(String args[])
{
String s="BEC" + "College";
System.out.println(s);
}
}
```
2. **By using concat() method:**
```
class x
{
public static void main(String args[])
{
String s1="Hi";
String s2="Hello";
String s3=s1.concat(s2);
System.out.println(s3);
}
}
```
3. **String concatenation with other data types**
```
Int age=19;
String str="He is" + age + "years old";
s.o.p(str);
```

Overriding toString() method

- The default toString() method from object Class prints "Class name @ hashcode".
- Default toString() method can be overridden

Using string methods:

```
class xx
{
public static void main(String args[])
{
String s ="Hello";
System.out.println(s.length());

char ch;
ch =s.charAt(2);
System.out.println(ch);


//substring
System.out.println("Substring:"
+s.substring(2,4));


//replace
String str;
str=s.replace('H','Y');
System.out.println(str);
```

RK KEYNOTES

Ex:
```
class xxx
{
int rollno;
String sname;

xxx(int rollno, String sname)
{
this.rollno=rollno;
this.sname=sname;
}

//public String toString() //override
//{
//return rollno + " " +name;
//}
public static void main(String args[])
{
xxx x1 = new xxx(101,"Ram");
System.out.println(x1); //xxx@3e25a5
}
}
```

RK KEYNOTES

Assignment:

Part A

1. **D/B method overloading and method overriding. 2m**
2. What is the use of super keyword? 2m
3. **D/B abstract class and concrete class 2m**
4. List abstract class rules 2m
5. What is final method? 2m
6. D/B superclass and subclass. 2m
7. **D/B extends and implements. 2m**
8. **D/B inheritance and polymorphism. 2m**
9. **D/B static and dynamic binding. 2m**
10. **Can abstract class be final? Why? 2m**
11. **Why is multiple inheritance using classes a disadvantage in java? 2m**
12. **D/B class and interface 2m**
13. **What is object cloning? 2m**
14. **D/B Array and ArrayList. 2m**
15. What is the use of iterator? 2m

16. List Drawbacks of ArrayList. How to overcome it? 2m
17. What is string? List few methods of string. 2m
18. **What is immutability in strings? 2m**
19. **What is collection framework? 2m**
20. List two ways to create strings with syntax. 2m

Part B

1. **Explain types of inheritance with neat diagram and example. 16m**
2. **What is constructor chaining? Example. 8m**
3. **Explain method overloading and method overriding with example. 16m**
4. Explain use of super keyword in different ways with example. 16m
5. Write example program for toString method using object class. 6m
6. **Explain abstract class with example.** 12m
7. Explain final method with example 12m
8. Is it possible extend interface? If so, prove with example. 16m
9. **Write a java program using object cloning concept. 16m**
10. What is inner class? List types with example. 16m
11. **What is ArrayList? Explain with example. 14m**
12. **Write an example program using string methods.** 16m
13. What are all the various ways to implement interface? 8m

RK KEYNOTES