RAJALAKSHMI INSTITUTE OF TECHNOLOGY

BELIEVE IN THE POSSIBILITIES

AN AUTONOMOUS INSTITUTION

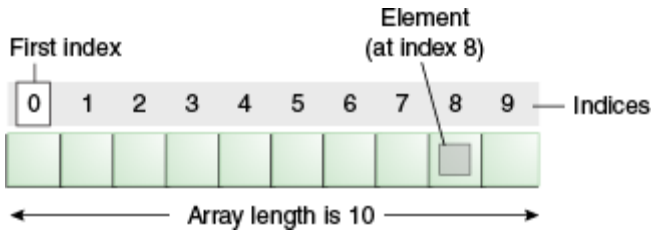
II Year/III Semester

CS23312- Object Oriented Programming

Arrays – Strings - Packages – Java-Doc comments –- Inheritance – class hierarchy –polymorphism – dynamic binding – final keyword – abstract classes-Exception handling – exception hierarchy – throwing and catching exceptions-The Object class – Reflection – interfaces – object cloning – inner classes – proxies - I/O Streams

2.1 ARRAYS

Array is a collection of similar type of elements that have contiguous memory location.
In Java all arrays are dynamically allocated.
Since arrays are objects in Java, we can find their length using member length.
A Java array variable can also be declared like other variables with [] after the data type.
The variables in the array are ordered and each have an index beginning from 0.
Java array can be also be used as a static field, a local variable or a method parameter.
The **size** of an array must be specified by an int value and not long or short.
The direct superclass of an array type is Object.
Every array type implements the interfaces Cloneable and java.io.Serializable.



Advantage of Java Array

Code Optimization: It makes the code optimized, we can retrieve or sort the data easily.
Random access: We can get any data located at any index position.

Disadvantage of Java Array

Size Limit: We can store only fixed size of elements in the array. It doesn't grow its size at runtime.To solve this problem, collection framework is used in java.

Types of Array in java

1. One- Dimensional Array
2. Multidimensional Array

One-Dimensional Arrays

An array is a group of like-typed variables that are referred to by a common name. An array

declaration has two components: the type and the name. *type* declares the element type of the array. The element type determines the data type of each element that comprises the array. We can also create an array of other primitive data types like char, float, double..etc or user defined data type(objects of a class).Thus, the element type for the array determines what type of data the array will hold.

Syntax:

type var-name[ ];

Instantiation of an Array in java

array-var = new type [size];

Example:

```
class Testarray{
public static void main(String args[]){
int a[]=new int[5];//declaration and instantiation
a[0]=10;//initialization
a[1]=20;
a[2]=70;
a[3]=40;
a[4]=50;
//printing array
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);
}}
```

Output:

```
10
20
70
40
50
```

Declaration, Instantiation and Initialization of Java ArrayExample:

```
class Testarray1{
public static void main(String args[]){
int a[]={33,3,4,5};//declaration, instantiation and initialization
//printing array
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);
}}
```

Output:

```
33
3
4
5
```

Passing Array to method in java

We can pass the java array to method so that we can reuse the same logic on any array.

Example:

```
class Testarray2{
static void min(int arr[]){
int min=arr[0];
for(int i=1;i<arr.length;i++)
 if(min>arr[i])
  min=arr[i];
 System.out.println(min);
}
public static void main(String args[]){
int a[]={33,3,4,5};
min(a);//passing array to method
}}
```

Output:

```
3
```

Multidimensional Arrays

Multidimensional arrays are arrays of arrays with each element of the array holding the reference of other array. These are also known as Jagged Arrays. A multidimensional array is created by appending one set of square brackets ([]) per dimension.

Syntax:

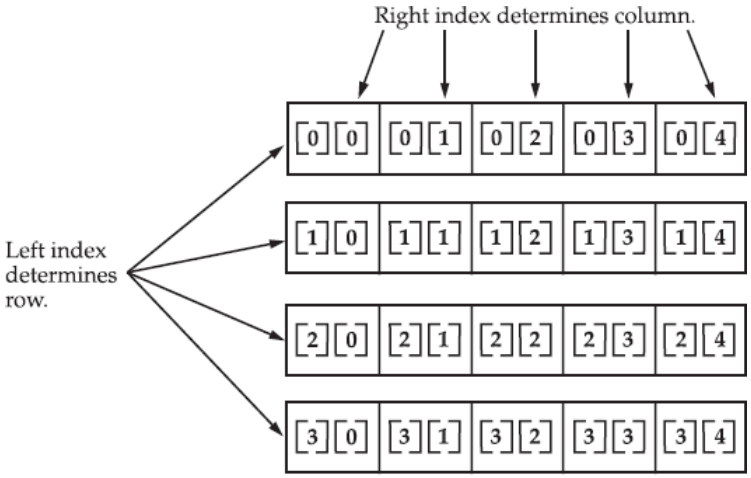
type var-name[ ][ ]=new type[row-size][col-size];

Example:

```
// Demonstrate a two-dimensional
array.class TwoDArray {
public static void main(String args[]) {
int twoD[][]= new int[4][5];
int i, j, k = 0;
for(i=0; i<4; i++)
for(j=0; j<5; j++) {
twoD[i][j] = k;
k++;
}
for(i=0; i<4; i++) {
for(j=0; j<5; j++)
System.out.print(twoD[i][j] + " ");
System.out.println();
}
}
}
```

Output:

```
0 1 2 3 4
5 6 7 8 9

10 11 12 13 14

15 16 17 18 19
```



Given: int twoD [ ] [ ] = new int [4] [5] ;

When you allocate memory for a multidimensional array, you need only specify the memory for the first (leftmost) dimension. You can allocate the remaining dimensions separately. For example, this following code allocates memory for the first dimension of **twoD** when it is declared. It allocates the second dimension manually.

Syntax:

```
int twoD[][] = new int[4][];
twoD[0] = new int[5];
twoD[1] = new int[5];
twoD[2] = new int[5];
twoD[3] = new int[5];
```

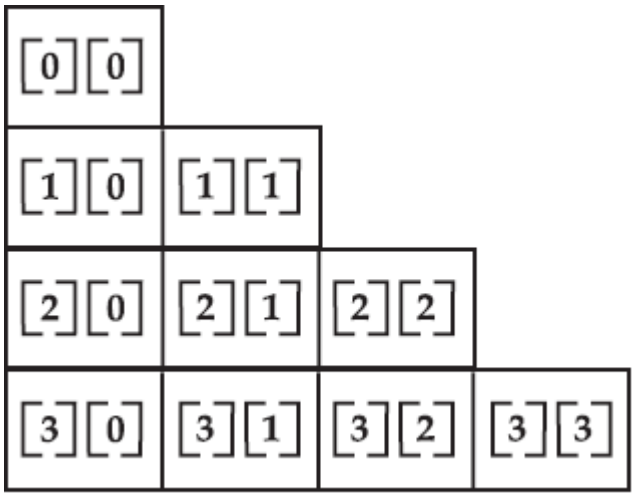
Example:

```
// Manually allocate differing size second
dimensions.class TwoDAgain {
public static void main(String args[])
{
int twoD[][] = new int[4][];
twoD[0] = new int[1];
twoD[1] = new int[2];
twoD[2] = new int[3];
twoD[3] = new int[4];
int i, j, k = 0;
for(i=0; i<4; i++)
for(j=0; j<i+1; j++)
{
twoD[i][j] = k;
k++;
}
for(i=0; i<4; i++)
{
for(j=0; j<i+1; j++)
System.out.print(twoD[i][j] + " ");

System.out.println();
}
}
}
```

Output:

0
1 2
3 4 5
6 7 8 9

The array created by this program looks like this:



JAVA DOC COMMENTS

The java comments are statements that are not executed by the compiler and interpreter. The comments can be used to provide information or explanation about the variable, method, class or any statement. It can also be used to hide program code for specific time.

There are 3 types of comments in java.

1. Single Line Comment
2. Multi Line Comment
3. Documentation Comment

1) Java Single Line Comment

The single line comment is used to comment only one line.

Syntax:

```
//This is single line comment
```

Example:

```
public class CommentExample1
{
 public static void main(String[] args)
 {
   int i=10;//Here, i is a
   variable
   System.out.println(i);
```

```
 }
}
```
Output:
10

2) Java Multi Line Comment

The multi line comment is used to comment multiple lines of code.

Syntax:

```
/*
Thisis
multi linecomment
*/
```
Example:
```
public class CommentExample2
{
  public static void main(String[] args)
  {
    /* Let's declare and
     print variable in
     java. */int i=10;
     System.out.println(
     i);
  }
}
```
Output:
10

3) Java Documentation Comment

The documentation comment is used to create documentation API. To create documentation API, you needto use **javadoc tool.**

Syntax:

```
    /** This is documentation comment */
```

Example:
```
/** The Calculator class provides methods to get addition and subtraction of given 2
numbers.*/

public class Calculator
{
/** The add() method returns addition of given
        numbers.*/
public static int add(int a, int b)
        {
        return a+b;
        }
/** The sub() method returns subtraction of given
        numbers.*/
public static int sub(int a, int b)
        {
        return a-b;
```

```
        }
}
```
This type of comment is used to produce an HTML file that documents your program. The documentationcomment begins with a /** and ends with a */

STRINGS

Basic String class

In java, string is basically an object that represents sequence of char values. Java String provides a lot

1. Compare String & StringBuffer Class. **(2) (NOV/DEC 2021)**
2. List the methods in String Class. **(2) (NOV/DEC 2021)**
3. Write a Java program to count the number of vowels in a given sentence. **(2) (NOV/DEC 2021)**

of concepts that can be performed on a string such as compare, concat, equals, split, length, replace, compareTo, intern, substring etc.

In java, string objects are immutable. Immutable simply means unmodifiable or unchangeable.

String s="javatpoint";

There are two ways to create String object:

- By string literal
- By new keyword

String Literal - Java String literal is created by using double quotes. For Example: String s="welcome";

By new keyword - String s=new String("Welcome");

Example 1: Create a String in Java

```
class Main {

  public static void main(String[] args) {

    // create strings

    String first = "Java";

    String second = "Python";

    String third = "JavaScript";

    // print strings

    System.out.println(first);  // print Java

    System.out.println(second); // print Python

    System.out.println(third);  // print JavaScript
```

```
  }
}
```

Creating strings using the new keyword

```
class Main {
  public static void main(String[] args) {
    // create a string using new
    String name = new String("Java String");
    System.out.println(name);  // print Java String
  }
}
```

String methods

String methods:		
1.	char charAt(int index)	returns char value for the particular index
2.	int length()	returns string length
3.	static String format(String format, Object... args)	returns formatted string
4.	static String format(Locale l, String format, Object... args)	returns formatted string with given locale
5.	String substring(int beginIndex)	returns substring for given begin index
6.	String substring(int beginIndex, int endIndex)	returns substring for given begin index and end Index
7.	boolean contains(CharSequence s)	returns true or false after matching the sequence of char value
8.	static String join(CharSequence delimiter, CharSequence... elements)	returns a joined string
9.	static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)	returns a joined string
10.	boolean equals(Object another)	checks the equality of string with object
11.	boolean isEmpty()	checks if string is empty
12.	String concat(String str)	concatinates specified string
13.	String replace(char old, char new)	replaces all occurrences of specified char value
14.	String replace(CharSequence old, CharSequence new)	replaces all occurrences of specified CharSequence
15.	static String equalsIgnoreCase(String another)	compares another string. It doesn't check case.
16.	String[] split(String regex)	returns splitted string matching regex
17.	String[] split(String regex, int limit)	returns splitted string matching regex and limit

18.	String intern()	returns interned string
19.	int indexOf(int ch)	returns specified char value index
20.	int indexOf(int ch, int fromIndex)	returns specified char value index starting with given index
21.	int indexOf(String substring)	returns specified substring index
22.	int indexOf(String substring, int fromIndex)	returns specified substring index starting with given index
23.	String toLowerCase()	returns string in lowercase.
24.	String toLowerCase(Locale l)	returns string in lowercase using specified locale.
25.	String toUpperCase()	returns string in uppercase.
26.	String toUpperCase(Locale l)	returns string in uppercase using specified locale.
27.	String trim()	removes beginning and ending spaces of this string.
28.	static String valueOf(int value)	converts given type into string. It is overloaded.

Example

```
public classstringmethod
{
public static void main(String[] args)
{
String string1 = new String("hello");
String string2 = new String("hello");
if (string1 == string2)
{
System.out.println("string1= "+string1+" string2= "+string2+" are equal");

}
else
{
System.out.println("string1= "+string1+" string2= "+string2+" are Unequal");
}
System.out.println("string1 and string2 is= "+string1.equals(string2));
String a="information";
System.out.println("Uppercase of String a is= "+a.toUpperCase());
String b="technology";
System.out.println("Concatenation of object a and b is= "+a.concat(b)); System.out.println("After concatenation Object a is= "+a.toString()); System.out.println("\"RIT\'s\" is the greatest\\ college in chennai"); System.out.println("Length of Object a is= "+a.length());
System.out.println("The third character of Object a is= "+a.charAt(2));

StringBuffer n=new StringBuffer("Technology");
StringBuffer m=new StringBuffer("Information");
```

```
System.out.println("Reverse of Object n is= "+n.reverse());
n= new StringBuffer("Technology");
System.out.println("Concatenation of Object m and n is= "+m.append(n)); System.out.println("After
concatenation of Object m is= "+m);
}
}
```

Output

```
string1= hello string2= hello are Unequal string1 and string2 is= true
Uppercase of String a is= INFORMATION
Concatenation of object a and b is= informationtechnology
After concatenation Object a is= information
"Joseph's" is the greatest\ college in chennai
Length of Object a is= 11
The third character of Object a is= f
Reverse of Object n is= ygolonhceT
Concatenation of Object m and n is= InformationTechnology
```

String Buffer Class

Java StringBuffer class is used to create mutable (modifiable) String objects. The StringBuffer class in Java is the same as String class except it is mutable i.e. it can be changed.

Important methods of StringBuffer class

Mutable String:

A String that can be modified or changed is known as mutable String. StringBuffer and StringBuilder classes are used for creating mutable strings.

StringBuffer Class append() Method

The append() method concatenates the given argument with this String.

Program

```
Class StringBufferExample
{
public static void main(String args[])
{

        StringBuffer sb=new StringBuffer("Hello ");
        sb.append("Compiler");//now original string is changed
        System.out.println(sb);//prints Hello Java
    }
    }
```

Output:

Hello Compiler

StringBuffer insert() Method:

The insert() method inserts the given String with this string at the given position.

Program:

```
class StringBufferExample2
{
public static void main(String args[])
{
StringBuffer sb=new StringBuffer("Hello ");
sb.insert(1,"Java");//now original string is changed
System.out.println(sb);//prints HJavaello
}
}
```

Output:

HJavaello

StringBuffer replace() Method

The replace() method replaces the given String from the specified beginIndex and endIndex.

StringBufferExample3.java

```
class StringBufferExample3{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello");
sb.replace(1,3,"Java");
System.out.println(sb);//prints HJavalo
}
}
```

OUTPUT:

```
HJavalo
```

StringBuffer delete() Method

The delete() method of the StringBuffer class deletes the String from the specified beginIndex to endIndex.

StringBufferExample4.java

```
class StringBufferExample4{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello");
sb.delete(1,3);
System.out.println(sb);//prints Hlo
}
}
```

OUTPUT

```
Hlo
```

StringBuffer reverse() Method

The reverse() method of the StringBuilder class reverses the current String.

StringBufferExample5.java

```
class StringBufferExample5{
```

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

StringBuffer sb=new StringBuffer("Hello");

sb.reverse();

System.out.println(sb);//prints olleH

} }
```

OUTPUT

OlleH

StringBuffer capacity() Method

The capacity() method of the StringBuffer class returns the current capacity of the buffer. The default capacity of the buffer is 16. If the number of character increases from its current capacity, it increases the capacity by (oldcapacity*2)+2. For example if your current capacity is 16, it will be (16*2)+2=34.

StringBufferExample6.java

```
class StringBufferExample6{

public static void main(String args[]){

StringBuffer sb=new StringBuffer();

System.out.println(sb.capacity());//default 16

sb.append("Hello");

System.out.println(sb.capacity());//now 16

sb.append("java is my favourite language");

System.out.println(sb.capacity());//now (16*2)+2=34 i.e (oldcapacity*2)+2

} }
```

OUTPUT

16

16 34

StringBuffer ensureCapacity() method

The ensureCapacity() method of the StringBuffer class ensures that the given capacity is the minimum to the

current capacity. If it is greater than the current capacity, it increases the capacity by (oldcapacity*2)+2. For example if your current capacity is 16, it will be (16*2)+2=34.

StringBufferExample7.java

```
class StringBufferExample7{
public static void main(String args[]){

StringBuffer sb=new StringBuffer();

System.out.println(sb.capacity());//default 16

sb.append("Hello");

System.out.println(sb.capacity());//now 16

sb.append("java is my favourite language");

System.out.println(sb.capacity());//now (16*2)+2=34 i.e (oldcapacity*2)+2

sb.ensureCapacity(10);//now no change

System.out.println(sb.capacity());//now 34

sb.ensureCapacity(50);//now (34*2)+2

System.out.println(sb.capacity());//now 70

} }
```

Output:

```
16

16

34

34

70
```

```
    System.out.println("success");

}

}
```

Output:

Success

testout.txt

Welcome to cse department

Java DataInputStream Class

Java DataInputStream class allows an application to read primitive data from the input stream in a machine-independent way.

Java application generally uses the data output stream to write data that can later be read by a data input stream.

Example of DataInputStream class

In this example, we are reading the data from the file testout.txt file.

```
import java.io.*;

public class DataStreamExample {

  public static void main(String[] args) throws IOException {

    InputStream input = new FileInputStream("D:\\testout.txt");

    DataInputStream inst = new DataInputStream(input);

    int count = input.available();

    byte[] ary = new byte[count];

    inst.read(ary);

    for (byte bt : ary) {

      char k = (char) bt;

      System.out.print(k+"-");

    }

  }

}
```

Here, we are assuming that you have following data in **"testout.txt"** file:

JAVA

Output:

J-A-V-A

Success

Java DataOutputStream Class

Java DataOutputStream class allows an application to write primitive Java data types to the output stream in a machine-independent way.

Java application generally uses the data output stream to write data that can later be read by a data input stream.

Java DataOutputStream class methods

Method	Description
int size()	It is used to return the number of bytes written to the data output stream.
void write(int b)	It is used to write the specified byte to the underlying output stream.
void write(byte[] b, int off, int len)	It is used to write len bytes of data to the output stream.
void writeBoolean(boolean v)	It is used to write Boolean to the output stream as a 1-byte value.
void writeChar(int v)	It is used to write char to the output stream as a 2-byte value.
void writeChars(String s)	It is used to write string to the output stream as a sequence of characters.
void writeByte(int v)	It is used to write a byte to the output stream as a 1-byte value.
void writeBytes(String s)	It is used to write string to the output stream as a sequence of bytes.
void writeInt(int v)	It is used to write an int to the output stream
void writeShort(int v)	It is used to write a short to the output stream.
void writeShort(int v)	It is used to write a short to the output stream.
void writeLong(long v)	It is used to write a long to the output stream.
void writeUTF(String str)	It is used to write a string to the output stream using UTF-8 encoding in portable manner.

void flush()	It is used to flushes the data output stream.

Example of DataOutputStream class

In this example, we are writing the data to a text file testout.txt using DataOutputStream class.

```
package com.javatpoint;

import java.io.*;
public class OutputExample {
   public static void main(String[] args) throws IOException {
      FileOutputStream file = new FileOutputStream(D:\\testout.txt);
      DataOutputStream data = new DataOutputStream(file);
      data.writeInt(65);
      data.flush();
      data.close();
      System.out.println("Succcess...");
   }
}
```

Output:

```
Succcess...
```

testout.txt:

```
A
```

Overloading Methods

1. Explain briefly about function overloading with a suitable example. Nov/Dec 2021

2. Outline Method overriding with an example. Nov/Dec 2021

When two or more methods within the same class that have the same name, but their parameter declarations are different. The methods are said to be overloaded, and the process is referred to as method overloading. Method overloading is one of the ways that Java supports polymorphism.

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

Example:

```
// Demonstrate method overloading. class OverloadDemo {

void test() {

System.out.println("No parameters");

}

// Overload test for one integer parameter. void test(int a) {

System.out.println("a: " + a);

}

// Overload test for two integer parameters. void test(int a, int b) {

System.out.println("a and b: " + a + " " + b);

}
```

```
// Overload test for a double parameter double test(double a) { System.out.println("double a: " + a); return a*a;

}

}

class Overload {

public static void main(String args[]) { OverloadDemo ob = new OverloadDemo(); double result;

// call all versions of test() ob.test();

ob.test(10);

ob.test(10, 20);

result = ob.test(123.25);

System.out.println("Result of ob.test(123.25): " + result);

}

}
```

Output:

```
No parameters

a: 10

a and b: 10 20

double a: 123.25

Result of ob.test(123.25): 15190.5625
```

Method Overriding

When a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to override the method in the superclass. When an overridden method is called from within its subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden.

Example:

```
// Method overriding. class A {

int i, j;

A(int a, int b) { i = a;

j = b;

}

// display i and j void show() {

System.out.println("i and j: " + i + " " + j);

}

}

class B extends A { int k;

B(int a, int b, int c) { super(a, b);

k = c;

}

// display k – this overrides show() in A void show() {

System.out.println("k: " + k);

}

}

class Override {
```

public static void main(String args[]) { B subOb = new B(1, 2, 3); subOb.show(); // this calls show() in B

}

}

Output:

k: 3

When show( ) is invoked on an object of type B, the version of show( ) defined within B is used. That is, the version of show( ) inside B overrides the version declared in A. If you wish to access the superclass version of an overridden method, you can do so by using super. For example, in this version of B, the superclass version of show( ) is invoked within the subclass' version. This allows all instance variables to be displayed. class B extends A {

int k;

B(int a, int b, int c) {

super(a, b); k = c;

}

void show() {

super.show(); // this calls A's show() System.out.println("k: " + k);

}

}

If you substitute this version of A into the previous program, you will see the following

Output:

i and j: 1 2

k: 3

Here, super.show( ) calls the superclass version of show( ).

1.1 Objects as Parameters – Returning Objects

Java does this interesting thing that's sort of a hybrid between pass-by-value and pass-by-reference. Basically, a parameter cannot be changed by the function, but the function can ask the parameter to change itself via calling some method within it.

- While creating a variable of a class type, we only create a reference to an object. Thus, when we pass this reference to a method, the parameter that receives it will refer to

the same object as that referred to by the argument.

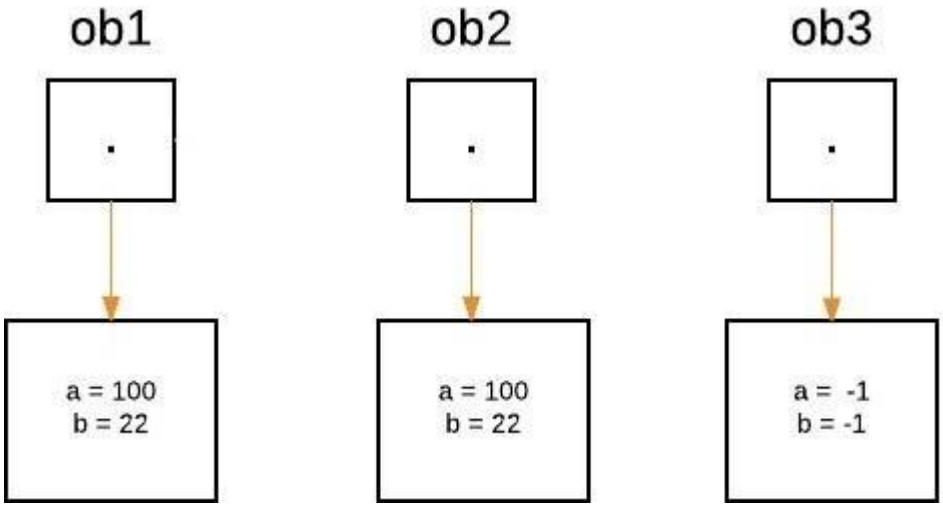
- This effectively means that objects act as if they are passed to methods by use of call - by-reference.
- Changes to the object inside the method do reflect the object used as an argument.

Illustration: Let us suppose three objects 'ob1' , 'ob2' and 'ob3' are created:

ObjectPassDemo ob1 = new ObjectPassDemo(100, 22);

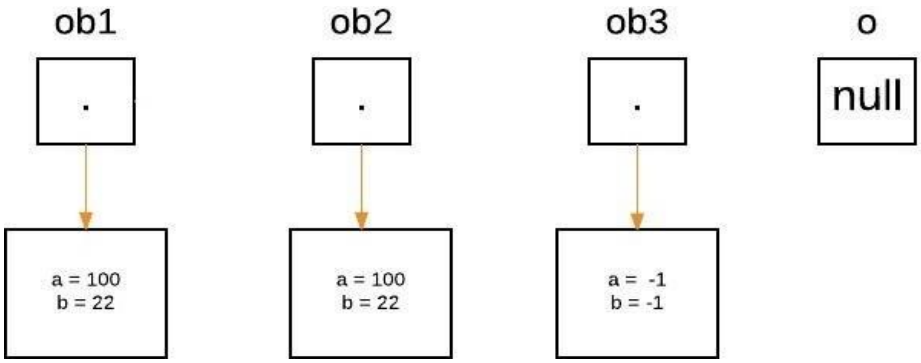
ObjectPassDemo ob2 = new ObjectPassDemo(100, 22);

ObjectPassDemo ob3 = new ObjectPassDemo(-1, -1);



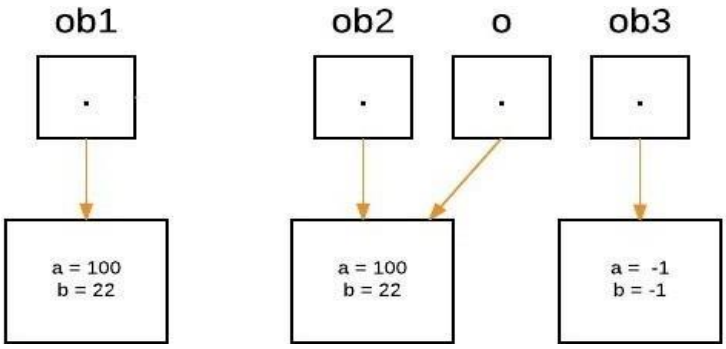
From the method side, a reference of type Foo with a name a is declared and it's initially assigned to null.

boolean equalTo(ObjectPassDemo o);



As we call the method equalTo, the reference 'o' will be assigned to the object which is passed as an argument, i.e. 'o' will refer to 'ob2' as the following statement execute.

System.out.println("ob1 == ob2: " + ob1.equalTo(ob2));

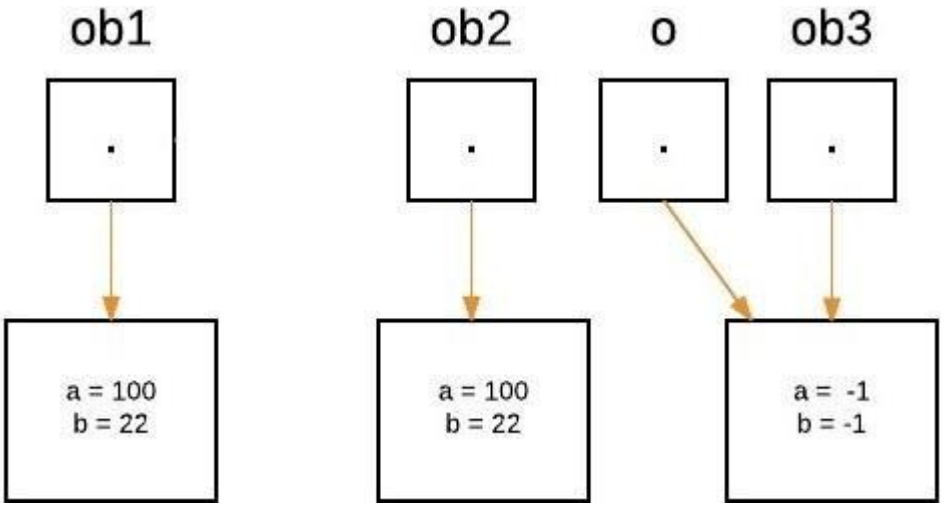


Now as we can see, equalTo method is called on 'ob1' , and 'o' is referring to 'ob2'. Since values of 'a' and 'b' are same for both the references, so if(condition) is true, so boolean true will be return.

if(o.a == a && o.b == b)

Again 'o' will reassign to 'ob3' as the following statement execute.

System.out.println("ob1 == ob3: " + ob1.equalTo(ob3));



- Now as we can see, the equalTo method is called on 'ob1', and 'o' is referring to 'ob3'. Since values of 'a' and 'b' are not the same for both the references, so if (condition) is false, so else block will execute, and false will be returned.

Example

```
// Java Program to Demonstrate Returning of Objects

// Class 1
class ObjectReturnDemo {
        int a;
```

```
        // Constructor
        ObjectReturnDemo(int i)
        { a = i; }

        // Method returns an object
        ObjectReturnDemo incrByTen()
        {
                ObjectReturnDemo temp = new ObjectReturnDemo(a + 10);
                return temp;
        }
}

// Class 2
// Main class
public class GFG {

        // Main driver method
        public static void main(String args[])
        {

                // Creating object of class1 inside main() method
                ObjectReturnDemo ob1 = new ObjectReturnDemo(2);
                ObjectReturnDemo ob2;

                ob2 = ob1.incrByTen();

                System.out.println("ob1.a: " + ob1.a);
                System.out.println("ob2.a: " + ob2.a);
        }
}
```

Output

ob1.a: 2

ob2.a: 12

Static, Nested and Inner Classes

Inner class means one class which is a member of another class. There are basically four types of inner classes in java.

1) Static nested classes
2) Nested Inner class
3) Method Local inner classes
4) Anonymous inner classes

1) Static nested classes

Static nested classes are not technically an inner class. They are like a static member of outer class.
Example:

```
class Outer {
private static void outerMethod() { System.out.println("inside outerMethod");
}
// A static inner class static class Inner {
public static void main(String[] args)
{
System.out.println("inside inner class Method");
outerMethod();
}
}
}
```

Output:
inside inner class Method inside outerMethod

2) Nested Inner class

Nested Inner class can access any private instance variable of outer class. Like any other instance variable, we can have access modifier private, protected, public and default modifier. Like class, interface can also be nested and can have access specifiers.

Example:

```
class Outer {

  // Simple
  nested inner
  class class
  Inner {
    public void show() {
        System.out.println("In a nested class method");
    }
  }
}
class Main {
  public static void
     main(String[] args) {
     Outer.Inner in = new
     Outer().new Inner();
```

```
    in.show();
  }
}
```

Output:

In a nested class method

3) Method Local inner classes

Inner class can be declared within a method of an outer class. In the following example, Inner is an inner class in outerMethod().

Example:

```
class Outer {
void outerMethod()
{
 System.out.println("inside  outerMethod");
// Inner class is local to outerMethod()
class Inner {
void innerMethod()
{
System.out.println("inside innerMethod");
}
}
Inner y = new Inner(); y.innerMethod();
}
}
class MethodDemo {
public static void main(String[] args) { Outer x = new Outer(); x.outerMethod();
}
}
```

Output:

Inside outerMethod Inside innerMethod

4) Anonymous inner classes

Anonymous inner classes are declared without any name at all. They are created in two ways.

a) As subclass of specified type

```
class Demo
{
void show() {
System.out.println("i am in show method of super class");
}
}
class Flavor1Demo {
```

```
// An anonymous class with Demo as base class static Demo d = new Demo() {
void show() { super.show();
System.out.println("i am in Flavor1Demo class");
}
};
public static void main(String[] args){ d.show();
}
}
```

Output:

i am in show method of super class i am in Flavor1Demo class

In the above code, we have two class Demo and Flavor1Demo. Here demo act as super class and anonymous class acts as a subclass, both classes have a method show(). In anonymous class show () method is overridden.

b) As implementer of the specified interface

Example:

```
class Flavor2Demo {
// An anonymous class that implements Hello interface static Hello h = new Hello() {
public void show() {
System.out.println("i am in anonymous class");
}
};
public static void main(String[] args) { h.show();
}
}
interface Hello { void show(); }
```

Output:

i am in anonymous class

In above code we create an object of anonymous inner class but this anonymous inner class is an implementer of the interface Hello. Any anonymous inner class can implement only one interface at one time. It can either extend a class or implement interface at a time.

INHERITANCE

1. Explain in detail about various types of inheritance in java with neat diagram? (NOV/DEC 2019)

2. Exemplify the use of super keyword. (NOV/DEC 2020)

3. Write a Java program to calculate electricity bill using inheritance. The program should get the inputs of watts per hour and unit rate. Check your program for the following case : Assume a consumer consumes 5000 watts per hour daily for One month. Calculate the total energy bill of that consumer if per unit rate is 7 [1 unit = 1k Wh]. (NOV/DEC 2020)

4. Outline the use of extends keyword in Java with syntax. (NOV/DEC 2021)

5. When a class hierarchy is created, in what order are the constructors for the classes that make up the hierarchy called? Outline with an example(NOV/DEC 2021)

6. Outline method overriding with an example. (NOV/DEC 2021)

Inheritance can be defined as the procedure or mechanism of acquiring all the properties and behaviors of one class to another, i.e., acquiring the properties and behavior of child class from the parent class.

When one object acquires all the properties and behaviours of another object, it is known as inheritance. Inheritance represents the **IS-A relationship**, also known as *parent-child* relationship.

Uses of inheritance in java

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

Types of inheritance in java: single, multilevel and hierarchical inheritance. Multiple and hybrid inheritance is supported through interface only.

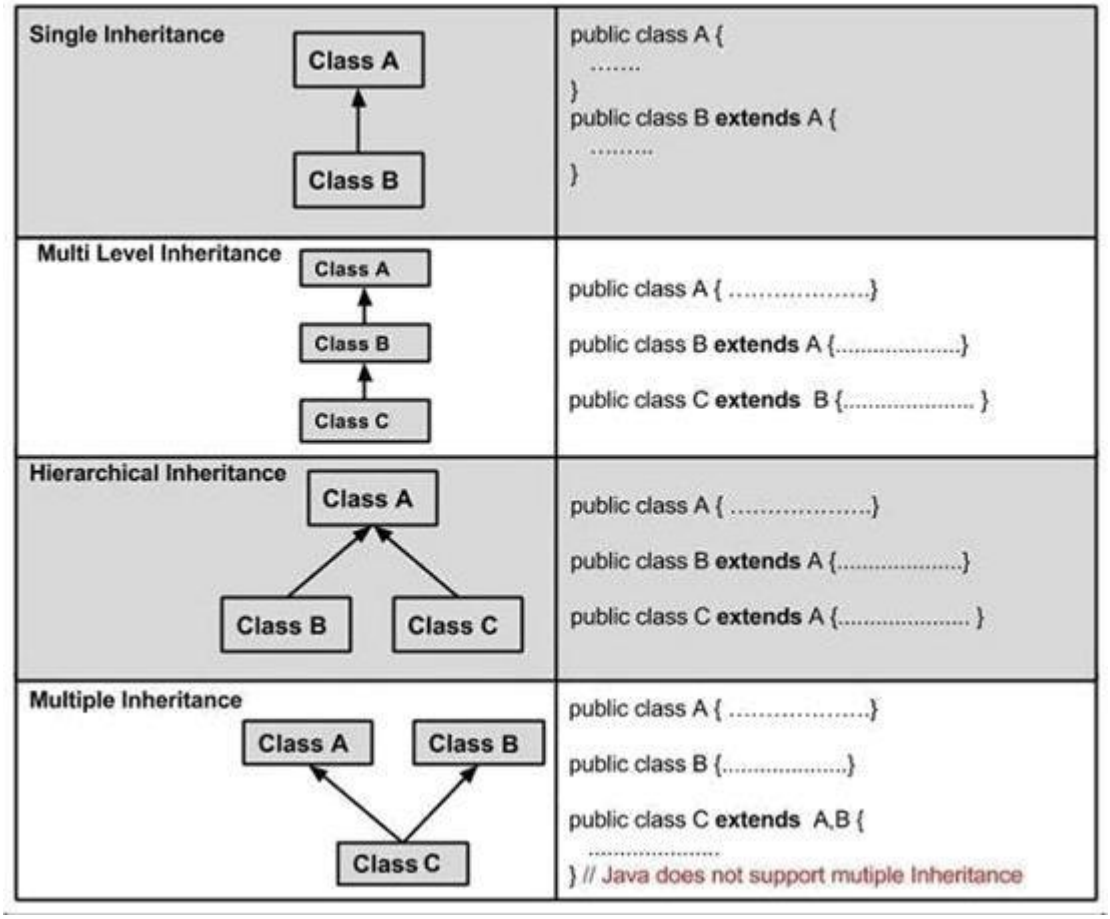
Syntax:

```
class subClass extends superClass
{
  //methods and fields
 }
```

Terms used in **Inheritence**

- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.

- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in previous class.



SINGLE INHERITANCE

In Single Inheritance one class extends another class (one class only).

Example:

```
public class ClassA
{
   public void dispA()
   {
      System.out.println("disp() method of ClassA");
   }
}
public class ClassB extends ClassA
```

```
{
    public void dispB()
    {
        System.out.println("disp() method of ClassB");
    }
    public static void main(String args[])
    {

        //Assigning ClassB object to
        ClassB reference ClassB b =
        new ClassB();
        //call dispA()
        method of
        ClassA
        b.dispA();
        //call dispB()
        method of
        ClassB
        b.dispB();
    }
}
```

Output :

```
disp() method of ClassA
disp() method of ClassB
```

MULTILEVEL INHERITANCE

In Multilevel Inheritance, one class can inherit from a derived class. Hence, the derived class becomes the base class for the new class.

Example:

```
public class ClassA
{
public void dispA()
{
System.out.println("disp() method of ClassA");
}
}
public class ClassB extends ClassA
{
public void dispB()
{
System.out.println("disp() method of ClassB");
}
}
```

```
public class ClassC extends ClassB
{
public void dispC()
{
System.out.println("disp() method of ClassC");
}
```

```
public static void main(String args[])
{
//Assigning ClassC object to ClassC reference ClassC c = new ClassC();
//call dispA() method of ClassA c.dispA();
//call dispB() method of ClassB c.dispB();
//call dispC() method of ClassC c.dispC();
}
}
```

Output :

disp() method of ClassA disp() method of ClassB disp() method of ClassC

HIERARCHICAL INHERITANCE

In Hierarchical Inheritance, one class is inherited by many sub classes.

Example:

```
public class ClassA
{
public void dispA()
{
System.out.println("disp() method of ClassA");
}
}
public class ClassB extends ClassA
{
public void dispB()
{
System.out.println("disp() method of ClassB");
}
}
public class ClassC extends ClassA
{
public void dispC()
{
System.out.println("disp() method of ClassC");
}
}
public class ClassD extends ClassA
{
public void dispD()
{
System.out.println("disp() method of ClassD");
}
}
```

```
public class HierarchicalInheritanceTest
{
public static void main(String args[])
{
//Assigning ClassB object to ClassB reference
ClassB b = new ClassB();
//call dispB() method of ClassB b.dispB();
//call dispA() method of ClassA b.dispA();
//Assigning ClassC object to ClassC reference ClassC c = new ClassC();
//call dispC() method of ClassC c.dispC();
//call dispA() method of ClassA c.dispA();

//Assigning ClassD object to ClassD reference ClassD d = new ClassD();
//call dispD() method of ClassD d.dispD();
//call dispA() method of ClassA d.dispA();
}
}
```

Output :

disp() method of ClassB disp() method of ClassA disp() method of ClassC disp() method of ClassA disp() method of ClassD disp() method of ClassA

Hybrid Inheritance is the combination of both Single and Multiple Inheritance. Again Hybrid inheritance is also not directly supported in Java only through interface we can achieve this. Flow diagram of the Hybrid inheritance will look like below. As you can ClassA will be acting as the Parent class for ClassB & ClassC and ClassB & ClassC will be acting as Parent for ClassD.

Multiple Inheritance is nothing but one class extending more than oneclass.Multiple Inheritance is basically not supported by many Object Oriented Programming languages such as Java, Small Talk, C# etc.. (C++SupportsMultiple Inheritance). As the Child class has to manage the dependency of more than one Parent class. But you can achieve multiple inheritance in Java using Interfaces.

"super" KEYWORD

Usage of super keyword

1. super() invokes the constructor of the parent class.
2. super.variable_name refers to the variable in the parent class.
3. super.method_name refers to the method of the parent class.

1. super() invokes the constructor of the parent class

super() will invoke the constructor of the parent class. Even when you don't add super() keyword the compiler will add one and will invoke the Parent Class constructor.

Example:

```
class ParentClass
{
ParentClass()
{
System.out.println("Parent Class default Constructor");
}
}
public class SubClass extends ParentClass
{
SubClass()
{
System.out.println("Child Class default Constructor");
}
public static void main(String args[])
{
SubClass s = new SubClass();
}
}
```

Output:

Parent Class default Constructor Child Class default Constructor

Even when we add explicitly also it behaves the same way as it did before.

```
class ParentClass
{
public ParentClass()
{
System.out.println("Parent Class default Constructor");
}
}
public class SubClass extends ParentClass
{
SubClass()
{
super();
System.out.println("Child Class default Constructor");
}
public static void main(String args[])
{
SubClass s = new SubClass();
}
}
```

Output:

Parent Class default Constructor

Child Class default Constructor

You can also call the parameterized constructor of the Parent Class. For example, super(10) will call parameterized constructor of the Parent class.

```
class ParentClass
{
ParentClass()
{
System.out.println("Parent Class default Constructor called");
}
ParentClass(int val)
{
System.out.println("Parent Class parameterized Constructor, value: "+val);
}
}
public class SubClass extends ParentClass
{
SubClass()
{
super();//Has to be the first statement in the constructor System.out.println("Child Class default Constructor called");
}
SubClass(int val)
{
super(10);
System.out.println("Child Class parameterized Constructor, value: "+val);
}
public static void main(String args[])
{
 }
}
```

Output

```
//Calling default constructor SubClass s = new SubClass();
//Calling parameterized constructor SubClass s1 = new SubClass(10);
```

```
Parent Class default Constructor called Child Class default Constructor called
Parent Class parameterized Constructor, value: 10
Child Class parameterized Constructor, value: 10
```

2. super.variable_name refers to the variable in the parent class

When we have the same variable in both parent and subclass class ParentClass

```
 {
    int val=999;
    }
    public class SubClass extends ParentClass
    {
    int val=123;

    void disp()
    {
    System.out.println("Value is : "+val);
    }

    public static void main(String args[])
    {

SubClass s = new SubClass(); s.disp();

    }
    }
```

Output

```
Value is : 123
```

This will call only the val of the sub class only. Without super keyword, you cannot call the val
which is present in the Parent Class. class ParentClass

```
{
int val=999;
}
public class SubClass extends ParentClass
{
int val=123;

void disp()
{
System.out.println("Value is : "+super.val);
}

public static void main(String args[])
{
SubClass s = new SubClass(); s.disp();
}
}
```

Output
Value is : 999

3. super.method_nae refers to the method of the parent class

When you override the Parent Class method in the Child Class without super keywords support you will not be able to call the Parent Class method. Let's look into the below example

```
class ParentClass
{
void disp()
{
System.out.println("Parent Class method");
}
}
public class SubClass extends ParentClass
{

void disp()
{
System.out.println("Child Class method");
}

void show()
{
disp();
}
public static void main(String args[])
{
SubClass s = new SubClass(); s.show();

}
}
```

Output:

Child Class method

Here we have overridden the Parent Class disp() method in the SubClass and hence SubClass disp() method is called. If we want to call the Parent Class disp() method also means then we have to use the super keyword for it.

```
class ParentClass
{
void disp()
```

```
    {
    System.out.println("Parent Class method");
    }
    }
    public class SubClass extends ParentClass
{
    void disp()
    {
    System.out.println("Child Class method");
    }

    void show()
    {
    //Calling SubClass disp() method disp();
    //Calling ParentClass disp() method super.disp();
    }
    public static void main(String args[])
    {

    }
    }
```

Output

```
    SubClass s = new SubClass(); s.show();
```

Child Class method Parent Class method

When there is no method overriding then by default Parent Class disp() method will be called.

```
    class ParentClass
    {
    public void disp()
    {
    System.out.println("Parent Class method");
    }
    }
    public class SubClass extends ParentClass
    {
    public void show()
    {
    disp();
    }
    public static void main(String args[])
    {
    SubClass s = new SubClass(); s.show(); }}
```

Output:
Parent Class method

Method Overriding

1. **Outline method overriding with an example. (NOV/DEC 2021)**

When a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to override the method in the superclass. When an overridden method is called from within its subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden.
Example:

```
// Method overriding. class A {
int i, j;
A(int a, int b) { i = a;
j = b;
}
// display i and j void show() {
System.out.println("i and j: " + i + " " + j);
}
}
class B extends A { int k;
B(int a, int b, int c) { super(a, b);
k = c;
}
// display k – this overrides show() in A void show() {
System.out.println("k: " + k);
}
}
class Override {
public static void main(String args[]) { B subOb = new B(1, 2, 3); subOb.show(); // this
calls show() in B
}
}
```

Output:
k: 3
When show( ) is invoked on an object of type B, the version of show( ) defined within B is used. That is, the version of show( ) inside B overrides the version declared in A. If you wish to access the superclass version of an overridden method, you can do so by using super. For example, in this version of B, the superclass version of show( ) is invoked within the subclass' version. This allows all instance variables to be displayed. class B extends A {

```
int k;
B(int a, int b, int c) {

super(a, b); k = c;
}
void show() {
super.show(); // this calls A's show() System.out.println("k: " + k);
}
}
```

If you substitute this version of A into the previous program, you will see the following Output:

```
i and j: 1 2
k: 3
```

Here, super.show( ) calls the superclass version of show( ).

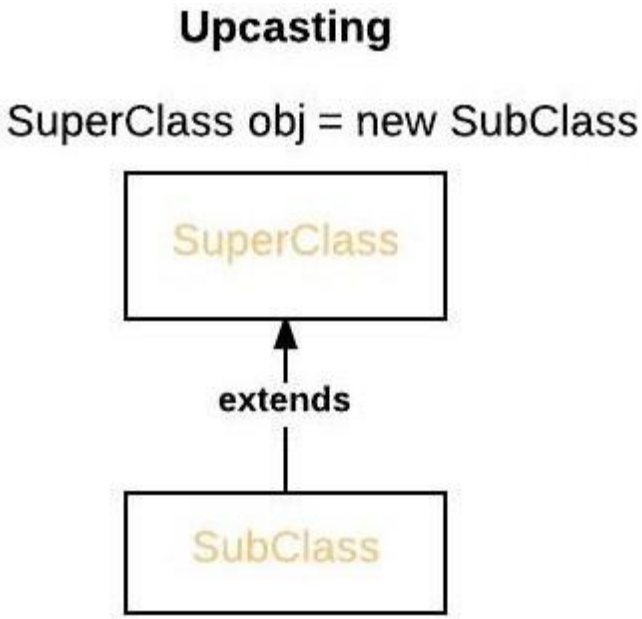
2.2 Dynamic Method Dispatch

Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.

When an overridden method is called through a superclass reference, Java determines which version(superclass/subclasses) of that method is to be executed based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time.

At run-time, it depends on the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed

A superclass reference variable can refer to a subclass object. This is also known as upcasting. Java uses this fact to resolve calls to overridden methods at run time.



Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed. Here is an example that illustrates dynamic method dispatch:

```
// A Java program to illustrate Dynamic Method
// Dispatch using hierarchical inheritance
class A
{
 void m1()
 {
  System.out.println("Inside A's m1 method");
 }
}

class B extends A
{
 // overriding m1()
 void m1()
 {
  System.out.println("Inside B's m1 method");
 }
}

class C extends A
{
 // overriding m1()
 void m1()
 {
  System.out.println("Inside C's m1 method");
 }
}

// Driver class
class Dispatch
{
 public static void main(String args[])
 {
  // object of type A
  A a = new A();

  // object of type B
```

```
        B b = new B();

        // object of type C
        C c = new C();

        // obtain a reference of type A
        A ref;

        // ref refers to an A object
        ref = a;

        // calling A's version of m1()
        ref.m1();

        // now ref refers to a B object
        ref = b;

        // calling B's version of m1()
        ref.m1();

        // now ref refers to a C object
        ref = c;

        // calling C's version of m1()
        ref.m1();
      }
}
```

Output:

Inside A's m1 method

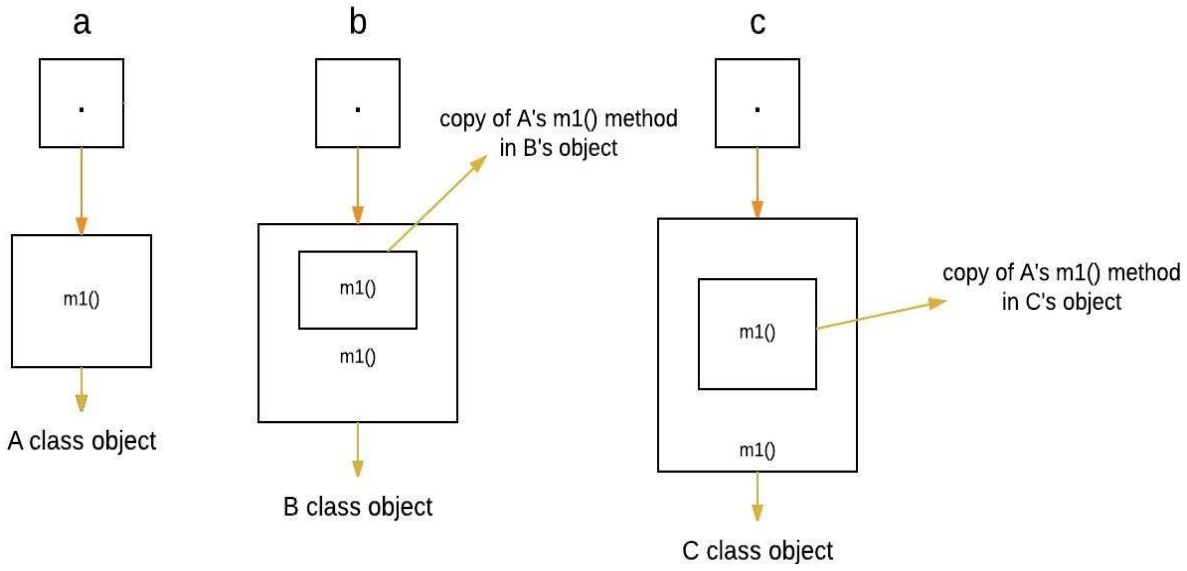
Inside B's m1 method

Inside C's m1 method

Explanation:

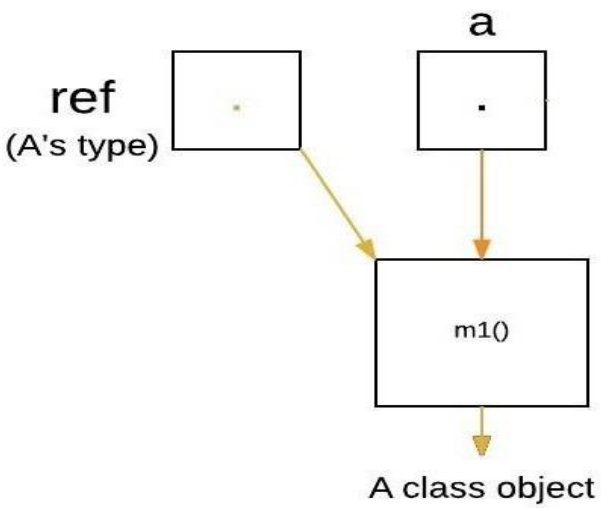
The above program creates one superclass called A and it's two subclasses B and C. These subclasses overrides m1( ) method.

1. Inside the main() method in Dispatch class, initially objects of type A, B, and C are declared.
2. A a = new A(); // object of type A
3. B b = new B(); // object of type B
4. C c = new C(); // object of type C



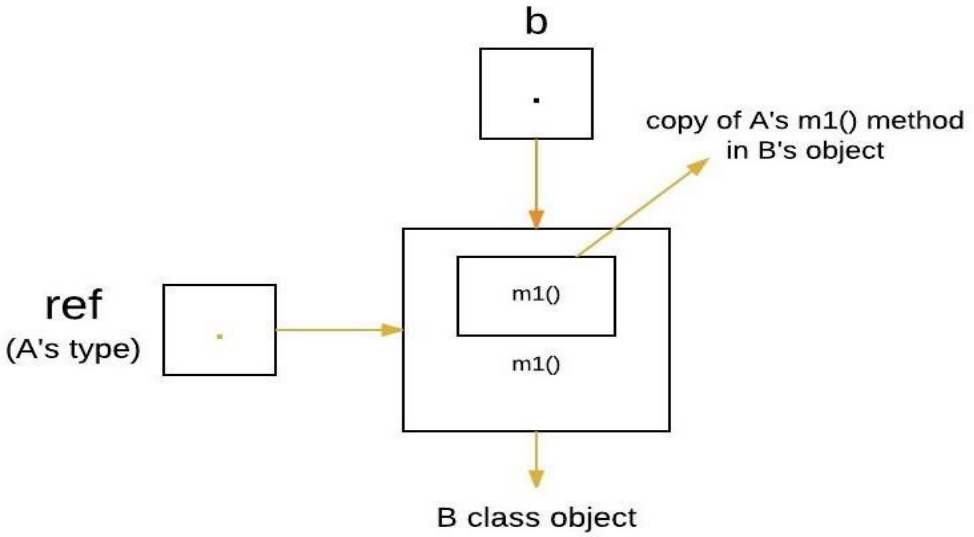
5. Now a reference of type A, called ref, is also declared, initially it will point to null.
6. A ref; // obtain a reference of type A

7. Now we are assigning a reference to each **type of object** (either A's or B's or C's) to *ref*, one-by-one, and uses that reference to invoke m1( ). As the output shows, the version of m1( ) executed is determined **by the type of object being referred to at the time of the call.**
8. ref = a; // r refers to an A object
9. ref.m1(); // calling A's version of m1()



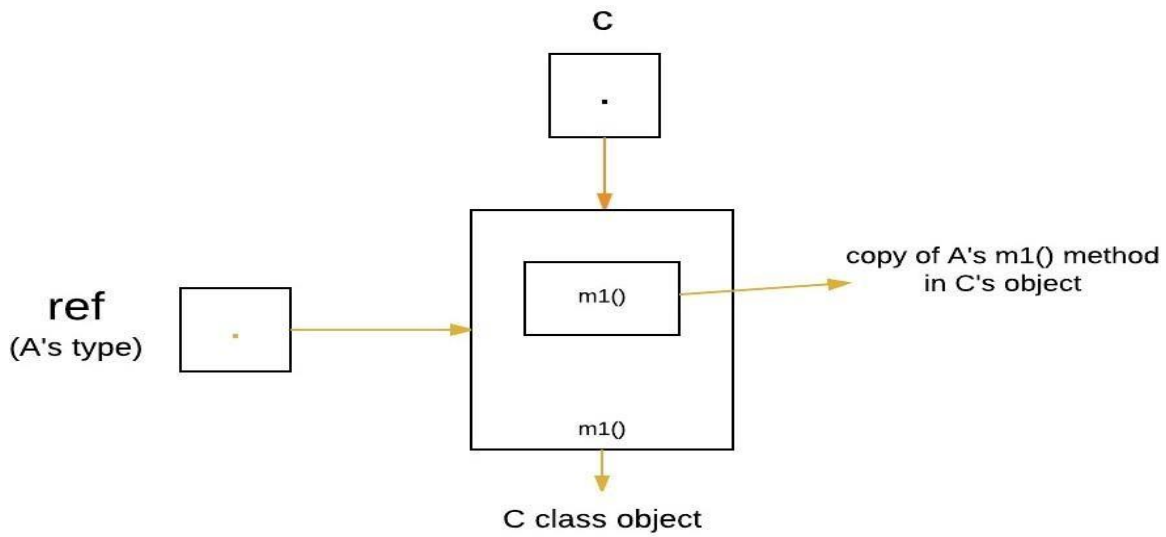
ref = b; // now r refers to a B object

ref.m1(); // calling B's version of m1()



ref = c; // now r refers to a C object

ref.m1(); // calling C's version of m1()



ABSTRACT CLASS

1. **What is an abstract class? Illustrate with an example to demonstrate abstract class? (NOV/DEC 2019)**
2. **When a class must be declared as abstract? (NOV/DEC 2021)**

A class that is declared with abstract keyword, is known as abstract class in java. It can have abstract and non-abstract methods (method with body).Abstraction is a process of hiding the implementation details and showing only functionality to the user.
Abstraction lets you focus on what the object does instead of how it does it. It needs to be extended and its method implemented. It cannot be instantiated.
Example abstract class:
abstract class A{}
abstract method:

A method that is declared as abstract and does not have implementation is known as abstract method.
abstract void printStatus();//no body and abstract

In this example, Shape is the abstract class, its implementation is provided by the Rectangle and Circle classes.
If you create the instance of Rectangle class, draw() method of Rectangle class will be invoked.
Example1:
File: TestAbstraction1.java abstract class Shape{ abstract void draw();
}
//In real scenario, implementation is provided by others i.e. unknown by end user class Rectangle extends Shape{

```
void draw(){System.out.println("drawing rectangle");}
}
class Circle1 extends Shape{
void draw(){System.out.println("drawing circle");}
}
//In real scenario, method is called by programmer or user class TestAbstraction1{
public static void main(String args[]){
Shape s=new Circle1();//In real scenario, object is provided through method e.g.
getShape() met hod
s.draw();
}
}
```

Output:

drawing circle

Abstract class having constructor, data member, methods

An abstract class can have data member, abstract method, method body, constructor and even main() method.

Example2:

File: TestAbstraction2.java

```
//example of abstract class that have method body abstract class Bike{
Bike(){System.out.println("bike is created");} abstract void run();
void changeGear(){System.out.println("gear changed");}
}
class Honda extends Bike{
void run(){System.out.println("running safely..");}
}

class TestAbstraction2{
public static void main(String args[]){ Bike obj = new Honda();
obj.run(); obj.changeGear();
}
}
```

Output:

bike is created running safely.. gear changed

The abstract class can also be used to provide some implementation of the interface. In such case, the end user may not be forced to override all the methods of the interface.

Example3:

```
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[]){ 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

Final with Inheritance

FINAL KEYWORD

Final keyword can be used along with variables, methods and classes.

1) final variable

2) final method

3) final class

1. Java final variable

A final variable is a variable whose value cannot be changed at anytime once assigned, it remains as a constant forever.

Example:

```
public class Travel
{
final int SPEED=60;
void increaseSpeed(){
SPEED=70;
}
public static void main(String args[])
{
Travel t=new Travel();
t.increaseSpeed();
}
}
```

Output :

Exception in thread "main" java.lang.Error: Unresolved compilation problem:

The final field Travel.SPEED cannot be assigned

The above code will give you Compile time error, as we are trying to change the value of a final
variable ‘SPEED’.

2. Java final method

When you declare a method as final, then it is called as final method. A final method cannot be overridden.

```
package com.javainterviewpoint;

class Parent
{
public final void disp()
{
System.out.println("disp() method of parent class");
}
}
public class Child extends Parent
{
public void disp()
{
System.out.println("disp() method of child class");
}
public static void main(String args[])
{
Child c = new Child(); c.disp();
}
}
```

Output : We will get the below error as we are overriding the disp() method of the Parent class. Exception in thread "main" java.lang.VerifyError: class
com.javainterviewpoint.Child overrides final method disp.()
at java.lang.ClassLoader.defineClass1(Native Method) at
java.lang.ClassLoader.defineClass(Unknown Source)
at java.security.SecureClassLoader.defineClass(Unknown Source) at
java.net.URLClassLoader.defineClass(Unknown Source)
at java.net.URLClassLoader.access$100(Unknown Source) at
java.net.URLClassLoader$1.run(Unknown Source)
at java.net.URLClassLoader$1.run(Unknown Source)

3. Java final class

A final class cannot be extended(cannot be subclassed), lets take a look into the below example package com.javainterviewpoint;

```
final class Parent
{
}
```

```
public class Child extends Parent
{
public static void main(String args[])
{
Child c = new Child();
}
}
```

Output :

We will get the compile time error like “The type Child cannot subclass the final class Parent”

Exception in thread "main" java.lang.Error: Unresolved compilation problem

EXCEPTION HANDLING BASICS

1. **What happens when the statement: int value = 25/0; is executed? (MAY/JUN 2016, APR/MAY 2017, APR/MAY 2019)**
2. **Define runtime exceptions. (NOV/DEC 2018)**
3. **What is the use of assert keyword? (NOV/DEC 2018, NOV/DEC 2018)**
4. **Explain the types of exceptions in java. (NOV/DEC 2019)**
5. **Write a Java program to create user define exception. (APR/MAY 2017)**
6. **What is exception handling? Explain with an example exception handling in java. (APR/MAY 2018)**
7. **Explain ‘Divide by zero’ Exception with an example. Java program. (APR/MAY 2019)**
8. **State the use of try block in Java exception handling. (NOV/DEC 2017)**
9. **Throw an exception if the matrices cannot be multiplied and handle it using a user defined exception. (NOV/DEC 2017)**
10. **Discuss in detail on how exceptions are handled with an appropriate example. (NOV/DEC 2019)**

1.1 Difference between error and exception

Errors indicate serious problems and abnormal conditions that most applications should not try to handle. Error defines problems that are not expected to be caught under normal circumstances by our program. For example memory error, hardware error, JVM error etc.

Exceptions are conditions within the code. A developer can handle such conditions and take necessary corrective actions. Few examples

- DivideByZero exception
- NullPointerException
- ArithmeticException
- ArrayIndexOutOfBoundsException

An exception (or exceptional event) is a problem that arises during the execution of a program. When an Exception occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled. If an exception is raised, which has not been handled by programmer then program execution can get terminated and system prints a non-user-friendly error message.

Ex: Exception in thread "main"
java.lang.ArithmeticException: / by zero at ExceptionDemo.main
(ExceptionDemo.java:5)
ExceptionDemo :
The class name main : The method
nameExceptionDemo.java : The
filename
java: 5: Line number

An exception can occur for many different reasons. Following are somescenarios where an exception occurs.

- A user has entered an **invalid data.**
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications orthe JVM has run out of memory.

Exception Hierarchy

All exception classes are subtypes of the java.lang.Exception class. Theexception class is a subclassof the Throwable class.

Keywords used in Exception handling

There are 5 keywords used in java exception handling.

1. try	A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code.
2. catch	A catch statement involves declaring the type of exception we are trying to catch.
3. finally	A finally block of code always executes, irrespective of occurrence of an Exception.
4. throw	It is used to execute important code such as closing connection, stream etc. throw is used to invoke an exception explicitly.
5. throws	throws are used to postpone the handling of a checked exception.

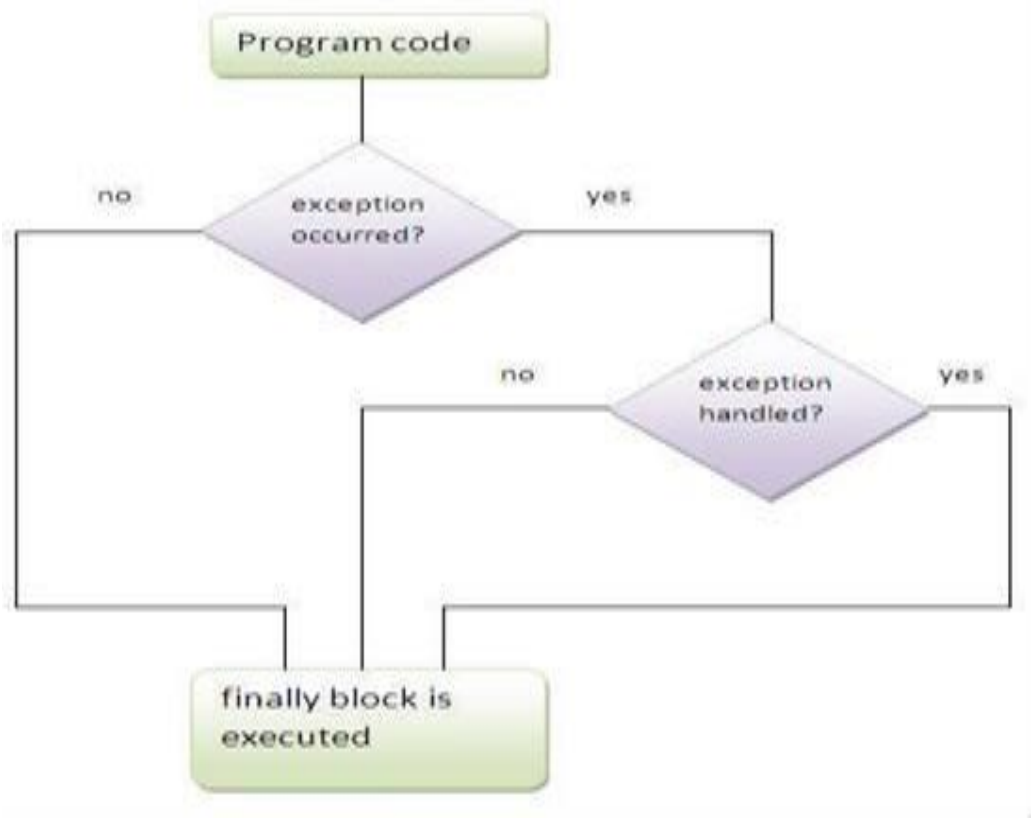
Syntax :

```
try
{
//Protected code
}

catch(ExceptionType1 e1)
{
//Catch block
}

catch(ExceptionType2 e2)
{
//Catch block
}
catch(ExceptionType3 e3)
{
//Catch block
}

finally
{
//The finally block always executes.
}
```

```
//Example-predefined Exception - for
//ArrayindexoutofBounds Exception public

class ExcepTest
{
public static void main(String args[])
{
int a[] = new int[2];
try
{
System.out.println("Access element three :" + a[3]);
}
catch(ArrayIndexOutOfBoundsException e)
{
 System.out.println("Exception thrown :" + e);
}

finally
{

a[0] = 6;

System.out.println("First element value: " + a[0]);

}

System.out.println("The finally statement is executed");
}
}
```

Output

```
Exception thrown :
java.lang.ArrayIndexOutOfBoundsException:3
First element value: 6
The finally statement is executed
```



Uncaught Exceptions

This small program includes an expression that intentionally causes a divide-by- zero error:class Exc0 {

```
public static void main(String args[])
{
int d = 0;
int a = 42 / d;
} }
```

When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then throws this exception. This causes the execution of Exc0 to stop, because once an exception has been thrown, it must be caught by an exception handler and dealt with immediately Any exception that is not caught by your program will ultimately be processed by the default handler.The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.

Here is the exception generated when this example is

executed: java.lang.ArithmeticException: / by zero at

Exc0.main(Exc0.java:4)

Stack Trace

Stack Trace is a list of method calls from the point when the application was started to the point where the exception was thrown. The most recent method calls are at the top. A stacktrace is avery helpful debugging tool.

when an Exception was thrown. This is very useful because it doesn't only show you where the error happened, but also how the program ended up in that place of the code. Using try and Catch to guard against and handle a run-time error, simply enclose the code that you want to monitor inside a try block. Immediately following the try block, include a catch clause that specifies the exception type that you wish to catch. A try and its catch statement form a unit. The following program includes a try block and a catch clause that processes the ArithmeticException generated by the division-by-zero error

```
class Exc2
{
public static void main(String args[])
{
in
t
d,
a;
tr
y
{
// monitor a block of
 code.d = 0;
  a = 42 / d;
  System.out.println("This will not be printed.");
  } catch (ArithmeticException e)
  { // catch divide-by-zero error
  System.out.println("Division by
  zero.");
  }
  System.out.println("After catch statement.");
  }
  }
```

This program generates the following output: Division by zero. After catch statement. The call to println( ) inside the try block is never executed. Once an exception is thrown, program control transfers out of the try block into the catch block.

3. PACKAGES AND INTERFACES

1. Describe the uses of interfaces in java?(NOV/DEC 2019)

2. What are the differences between classes and interfaces ? (NOV/DEC 2020)

3. What is interface ? With an example explain how to define and implement interface. (NOV/DEC 2020)

4. Write a note on interfaces and present the syntax for defining an interface. (NOV/DEC 2021)

5. Outline how interfaces are implemented in java with an example? (NOV/DEC 2021)

A java package is a group of similar types of classes, interfaces and sub-packages. Package in java can be categorized in two form, built-in package and user-defined package. There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.
2) Java package provides access protection.
3) Java package removes naming collision.

Defining a Package

To create a package include a package command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package. The package statement defines a name space in which classes are stored. If package statement is omitted, the class names are put into the default package, which has no name.

Syntax:

package <fully qualified package name>;

package pkg;

Here, pkg is the name of the package. For example, the following statement creates a package called MyPackage.

package 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 possible to create a hierarchy of packages. The general form of a multileveled package statement is shown here:
package pkg1[.pkg2[.pkg3]];

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. We cannot rename a package without renaming the directory in which the classes are stored.

Finding Packages and CLASSPATH

First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in a subdirectory of the current directory, it will be found. Second, you can specify a directory path or paths by setting the CLASSPATH environmental variable. Third, you can use the - classpath option with java and javac to specify the path to your classes.

consider the following package specification:
package MyPack
In order for a program to find MyPack, one of three things must be true. Either the program can be executed from a directory immediately above MyPack, or the CLASSPATH must be set to include the path to MyPack, or the -classpath option must specify the path to MyPack when the program is run via java.

When the second two options are used, the class path must not include MyPack, itself. It must simply specify the path to MyPack. For example, in a Windows environment, if the path to MyPack is C:\MyPrograms\Java\MyPack then the class path to MyPack is C:\MyPrograms\Java

Example:

```
// A simple package package MyPack; class Balance { String name; double bal;
Balance(String n, double b) { name = n;
bal = b;
}
void show() { if(bal<0)
System.out.print("--> "); System.out.println(name + ": $" + bal);
}
}
class AccountBalance {
public static void main(String args[]) { Balance current[] = new Balance[3];
current[0] = new Balance("K. J. Fielding", 123.23); current[1] = new Balance("Will Tell",
157.02); current[2] = new Balance("Tom Jackson", -12.33); for(int i=0; i<3; i++)
current[i].show();
}
}
```

Call this file AccountBalance.java and put it in a directory called MyPack.
Next, compile the file. Make sure that the resulting .class file is also in the MyPack directory. Then, try executing the AccountBalance class, using the following command line:
java MyPack.AccountBalance
Remember, you will need to be in the directory above MyPack when you execute this command. (Alternatively, you can use one of the other two options described in the preceding section to specify the path MyPack.)
As explained, AccountBalance is now part of the package MyPack. This means that it cannot be executed by itself. That is, you cannot use this command line:

java AccountBalance
AccountBalance must be qualified with its package name.

Example: package pck1; class Student
{
private int rollno; private String name; private String address;
public Student(int rno, String sname, String sadd)
{
rollno = rno; name = sname; address = sadd;
}
public void showDetails()
{
System.out.println("Roll No :: " + rollno); System.out.println("Name :: " + name);
System.out.println("Address :: " + address);
}
}
public class DemoPackage
{
public static void main(String ar[])
{
Student st[]=new Student[2];

```
st[0] = new Student (1001,"Alice", "New York"); st[1] = new
Student(1002,"BOb","Washington"); st[0].showDetails();
st[1].showDetails();
}
}
```

There are two ways to create package directory as follows:

1. Create the folder or directory at your choice location with the same name as package name. After compilation of copy .class (byte code file) file into this folder.
2. Compile the file with following syntax.

javac -d <target location of package> sourceFile.java

The above syntax will create the package given in the sourceFile at the <target location of pacakge> if it is not yet created. If package already exist then only the .class (byte code file) will be stored to the package given in sourceFile.

Steps to compile the given example code:

Compile the code with the command on the command prompt. javac -d DemoPackage.java

1. The command will create the package at the current location with the name pck1, and contains the file DemoPackage.class and Student.class

2. To run write the command given below java pckl.DemoPackage

Note: The DemoPackate.class is now stored in pck1 package. So that we've to use fully qualified type name to run or access it.

Output:

```
Roll No :: 1001 Name :: Alice
Address :: New York Roll No ::
1002 Name :: Bob
Address :: Washington
```

Interfaces

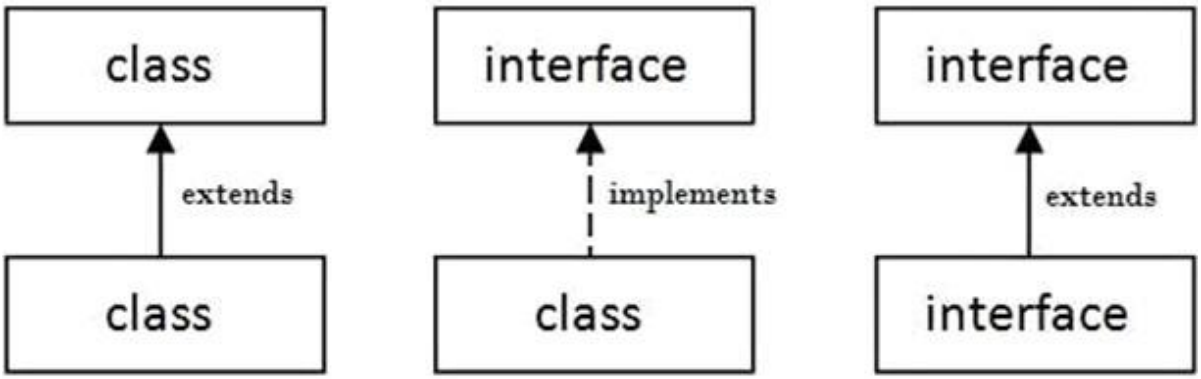
An interface in java is a blueprint of a class. It has static constants and abstract methods.The interface in java is a mechanism to achieve abstraction and multiple inheritance.

Interface is declared by using interface keyword. It provides total abstraction; means all the methods in interface are declared with empty body and are public and all fields are public, static and final by default. A class that implement interface must implement all the methods declared in the interface.

Syntax:
interface <interface_name>
{

// declare constant fields
// declare methods that abstract
// by default.
}

Relationship between classes and interfaces

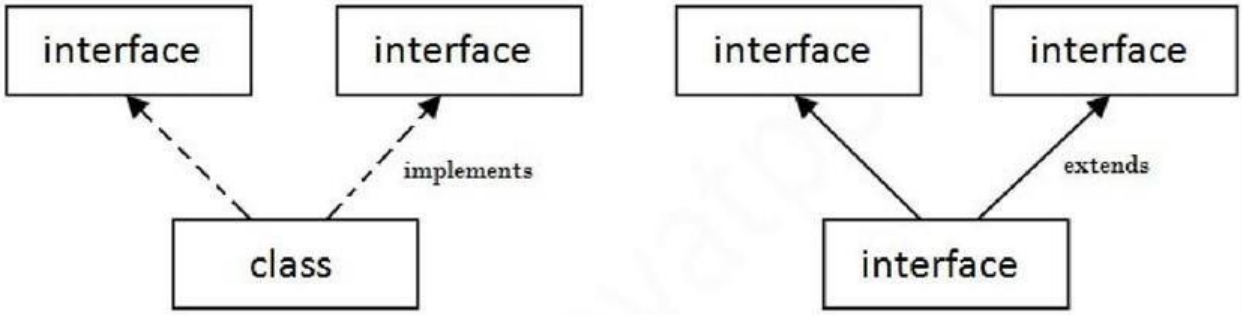


Example: interface printable{ void print();
}
class A6 implements printable{
public void print(){System.out.println("Hello");} public static void main(String args[]){ A6 obj = new A6(); obj.print();
}
}
Output:
Hello

Example: interface Drawable
{
void draw();
}
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}
class Circle implements Drawable{
public void draw(){System.out.println("drawing circle");}
}

```
class TestInterface1{
public static void main(String args[]){
Drawable d=new Circle();//In real scenario, object is provided by method e.g.
getDrawable() d.draw();
}
}
```
Output:
drawing circle

Multiple inheritance in Java by interface
If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.



Example: interface Printable{ void print();
}
interface Showable{ void show();
}
class A7 implements Printable,Showable{ public void
print(){System.out.println("Hello");}

public void show(){System.out.println("Welcome");} public static void main(String args[]){
A7 obj = new A7(); obj.print();
obj.show();
}
}
Output:
Hello Welcome

Interface inheritance

A class implements interface but one interface extends another interface . Example: interface Printable{ void print();

}
interface Showable extends Printable{ void show();
}
class TestInterface4 implements Showable{ public void
print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");} public static void main(String args[]){

```
TestInterface4 obj = new TestInterface4(); obj.print();
obj.show();
}}
```
Output:
Hello Welcome

Nested Interface in Java

An interface can have another interface i.e. known as nested interface.
```
interface printable{
void print();
interface MessagePrintable{ void msg();
}
}
```

Key points to remember about interfaces:

1) We can't instantiate an interface in java. That means we cannot create the object of an interface
2) Interface provides full abstraction as none of its methods have body. On the other hand abstract class provides partial abstraction as it can have abstract and concrete(methods with body) methods both.
3) "implements" keyword is used by classes to implement an interface.
4) While providing implementation in class of any method of an interface, it needs to be mentioned as public.
5) Class that implements any interface must implement all the methods of that interface, else the class should be declared abstract.
6) Interface cannot be declared as private, protected or transient.
7) All the interface methods are by default abstract and public.
8) Variables declared in interface are public, static and final by default.
```
interface Try
{
int a=10; public int a=10;
public static final int a=10; final int a=10; static int
a=0;
}
```
All of the above statements are identical.

9) Interface variables must be initialized at the time of declaration otherwise compiler will throw an error.
```
interface Try

int x;//Compile-time error
}
```
Above code will throw a compile time error as the value of the variable x is not initialized at the time of declaration.

10) Inside any implementation class, you cannot change the variables declared in interface because by default, they are public, static and final. Here we are implementing the interface "Try" which has a variable x. When we tried to set the value for variable x we got compilation error as the variable x is public static final by default and final variables can not be re-initialized.
```
class Sample implements Try
{
public static void main(String args[])
{
x=20; //compile time error
```

```
}
}
```
11) An interface can extend any interface but cannot implement it. Class implements interface and interface extends interface.

12) A class can implement any number of interfaces.

13) If there are two or more same methods in two interfaces and a class implements both interfaces, implementation of the method once is enough.

```
interface A
{
public void aaa();
}
interface B
{
public void aaa();
}
class Central implements A,B
{
public void aaa()
{
//Any Code here
}
public static void main(String args[])
{
//Statements
}
}
```
14) A class cannot implement two interfaces that have methods with same name but different return type.

```
interface A

public void aaa();
}
interface B
{
public int aaa();
}
class Central implements A,B
{
public void aaa() // error
{
}
public int aaa() // error
{
}
public static void main(String args[])
{
}
}
```
15) Variable names conflicts can be resolved by interface name.

```
interface A
```

```
{
int x=10;
}
interface B
{
int x=100;
}
class Hello implements A,B
{
public static void Main(String args[])
{
System.out.println(x); System.out.println(A.x);
System.out.println(B.x);
}
}
```

Advantages of interface in java:

- Without bothering about the implementation part, we can achieve the security of implementation
- In java, multiple inheritance is not allowed, however you can use interface to make use of it as you can implement more than one interface.

I/O STREAMS

1. What is InputStream? Present an outline of the methods defined by InputStream. **(NOV/DEC 2021)(13)**
2. How character streams are defined?(2) **(NOV/DEC 2021)(13)**
3. What are the uses of streams. What are the two types of streams ?**(NOV/DEC 2020)**
4. Explain in detail about reading and writing console in java with sample example program?

Java I/O (Input and Output) is used to process the input and produce the output based on the input. Java uses the concept of stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.

1.1 I/O Basics

A stream can be defined as a sequence of data. there are two kinds of Streams

- **InputStream:** The InputStream is used to read data from a source.
- **OutputStream:** The OutputStream is used for writing data to a destination.

Byte Streams

Java byte streams are used to perform input and output of 8-bit bytes FileInputStream , FileOutputStream.

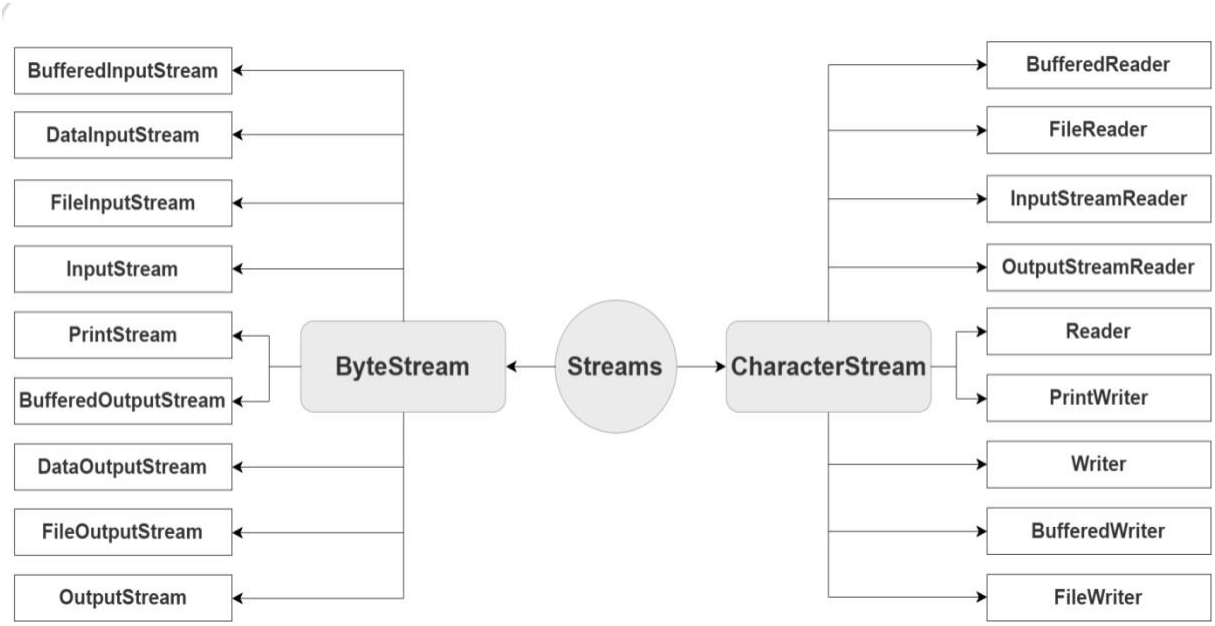
Character Streams

Java Character streams are used to perform input and output for 16-bit unicode. FileReader , FileWriter

Standard Streams

- **Standard Input:** This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as System.in.
- **Standard Output:** This is used to output the data produced by the user's program and usually a computer screen is used to standard output stream and represented as System.out.
- **Standard Error:** This is used to output the error data produced by the user's program and usually a computer screen is used to standard error stream and represented as System.err.

Classification of Stream Classes



Byte Stream Classes

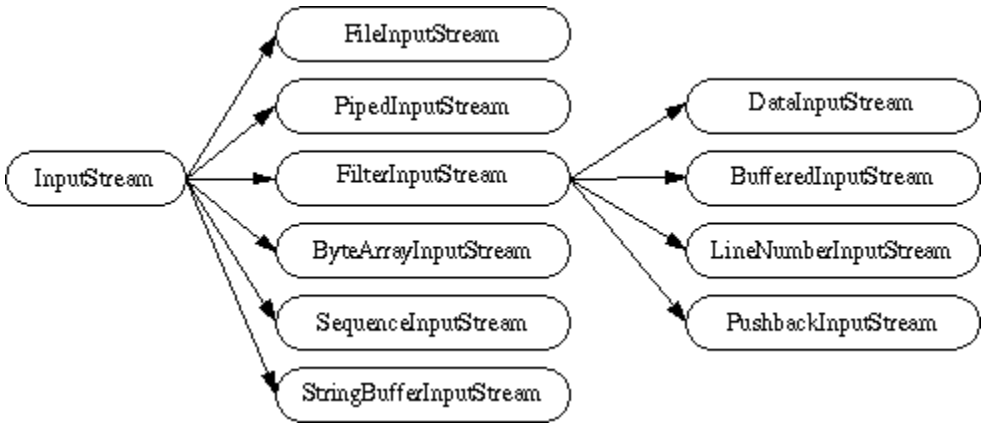
ByteStream classes have been designed to provide functional features for creating and manipulating streams and files for reading and writing bytes. Since the streams are unidirectional, they can transmit bytes in only one direction and therefore, Java provides two kinds of byte stream classes: InputStream class and OutputStream class.

Input Stream Classes

Input stream classes that are used to read 8-bit bytes include a super class known as InputStream and number of subclasses for supporting various input- related functions**.**

1.2 Reading and writing console I/O

Hierarchy of Input Stream Classes



The super class InputStream is an abstract class, so we cannot create object for the class. InputStream class defines the methods to perform the following functions:-

- Reading Bytes
- Closing Streams
- Marking position in Streams
- Skipping ahead in streams
- Finding the number of bytes in stream.

The following are the InputStream methods:

	Method	*Description*
1.	read( )	Reads a byte from the input stream
2.	read (byte b[ ])	Reads an array of bytes into b
3.	read (byte b[ ], int n, int m)	Reads m bytes into b starting from nth byte.
4.	available( )	Gives number of bytes available in the input
5.	skip(n)	Skips over n bytes from the input stream
6.	reset( )	Goes back to the beginning of the stream
7.	close( )	Closes the input stream

The DataInput interface contains the following methods

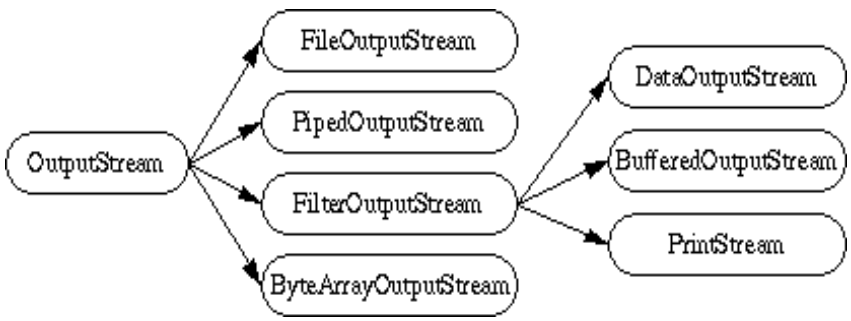
- readShort( )
- readInt( )
- readLong( )
- readFloat( )
- readUTF( )
- readDouble( )
- readLine( )
- readChar( )
- readBollean( )

OutputStream Class

The super class InputStream is an abstract class, so we cannot create object for the class. InputStream class defines the methods to perform the following functions:

- Writing Bytes
- Closing Streams
- Flushing Streams

Hierarchy of Output Stream Classes



OutputStream Method

Method	*Description*
1. write( )	Writes a byte to the output stream
2. write(byte[ ] b)	Writes all bytes in the array b to the output stream
3. write(byte b[ ], int n, int m)	Writes m bytes from array b starting from nth byte
4. close( )	Closes the output stream
5. flush( )	Flushes the output stream

Character Stream Vs Byte Stream in Java I/O Stream

A stream is a method to sequentially access a file. I/O Stream means an input source or output destination representing different types of sources e.g. disk files.The java.io package provides classes thatallow you to convert between Unicode character streams and byte streams of non-Unicode text.

Stream: A sequence of data.

Input Stream: reads data from

source. **Output Stream:** writes

data to destination.**Character**

Stream

In Java, characters are stored using Unicode conventions. Character stream automatically allows usto read/write data character by character. For example FileReader and FileWriter are character streams used to read from source andwrite to destination.

Java BufferedInputStream Class

Java BufferedInputStream class is used to read information from stream. It internally uses buffermechanism to make the performance fast.

The important points about BufferedInputStream are:

1. When the bytes from the stream are skipped or read, the internal buffer automatically refilled from thecontained input stream, many bytes at a time.

2. When a BufferedInputStream is created, an internal buffer array is created.

Example of Java BufferedInputStream

```
import java.io.*;
public class
 BufferedInputStreamExample{
 public static void main(String
 args[]){
  try{
    FileInputStream fin=new
    FileInputStream("D:\\testout.txt");
    BufferedInputStream bin=new
    BufferedInputStream(fin);
    int i;
    while((i=bin.read())!=-1){System.out.print((char)i);
    }
    bin.close();
    fin.close();
  }catch(Exception e){System.out.println(e);}
 }
}
```

Java BufferedOutputStream Class

Java BufferedOutputStream class is used for buffering an output stream. It internally uses buffer to storedata. It adds more efficiency than to write data directly into a stream. So, it makes the performance fast.

For adding the buffer in an OutputStream, use the BufferedOutputStream class. Let's see the syntax foradding the buffer in an OutputStream:

OutputStream os= new BufferedOutputStream(new FileOutputStream("D:\\IO Package\\testout.txt"));

Example of Java BufferedOutputStream

In this example, we are writing the textual information in the BufferedOutputStream object which is connected to the FileOutputStream object. The flush() flushes the data of one stream and send it into another. It is required if you have connected the one stream with another.

```
import java.io.*;

public class BufferedOutputStreamExample{

public static void main(String args[])throws Exception{

    FileOutputStream fout=new FileOutputStream("D:\\testout.txt");

    BufferedOutputStream bout=new BufferedOutputStream(fout);

    String s="Welcome to cse department";

    byte b[]=s.getBytes();bout.write(b); bout.flush();

    bout.close();

    fout.close();
```

Object Class

In Java, the `Object` class is the root class for all Java classes. This means that every class in Java, either directly or indirectly, inherits from the `Object` class. Because of this, the `Object` class provides a set of methods that are available to all Java objects, regardless of their specific class.

Here are some key points about the `Object` class:

1. **Common Methods**: The `Object` class defines several methods that are available to all objects in Java. Some of the most commonly used methods include:

- `toString()`: Returns a string representation of the object.
- `equals(Object obj)`: Compares the current object with another object for equality.
- `hashCode()`: Returns a hash code value for the object.
- `getClass()`: Returns the runtime class of the object.
- `clone()`: Creates and returns a copy of the object (requires the class to implement `Cloneable`).
- `notify()`, `notifyAll()`, and `wait()`: These methods are used for thread synchronization.

2. **Inheritance**: Since every class inherits from `Object`, it can override methods from `Object` to provide more specific behavior. For example, you might override the `toString()` method to provide a more meaningful string representation of your object.
3. **Usage**: The `Object` class methods can be particularly useful for generic programming. For instance, when you work with collections or other APIs that use `Object`, you often rely on these methods.
4. **Constructor**: The `Object` class has a default constructor that is implicitly called when a class is instantiated.

Here's a brief example demonstrating how you might override some of these methods in a custom class:

```
java
Copy code
public class MyClass {
    private int id;
    private String name;

    public MyClass(int id, String name) {
        this.id = id;
        this.name = name;
    }

    @Override
    public String toString() {
        return "MyClass{id=" + id + ", name='" + name + "'}";
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;
        MyClass myClass = (MyClass) obj;
        return id == myClass.id && name.equals(myClass.name);
    }

    @Override
    public int hashCode() {
        int result = Integer.hashCode(id);
        result = 31 * result + name.hashCode();
        return result;
    }
}
```

In this example, the `toString()`, `equals()`, and `hashCode()` methods are overridden to provide custom behavior specific to the `MyClass` class.

Reflection

In Java, reflection is a powerful feature that allows you to inspect and manipulate classes, methods, fields, and other components of the Java language at runtime. It's part of the `java.lang.reflect` package and provides a way to interact with Java objects and classes in a dynamic and flexible manner.

Here's a high-level overview of what reflection allows you to do:

1. **Inspecting Classes**: Reflection lets you obtain information about a class, such as its name, superclass, implemented interfaces, methods, fields, and constructors. This can be done using the `Class` class and its various methods, like `getName()`, `getSuperclass()`, and `getInterfaces()`.
2. **Instantiating Objects**: You can create new instances of classes dynamically using reflection. This is done via the `Class` object's `newInstance()` method (or the more recent `getDeclaredConstructor().newInstance()` method in Java 9 and later).
3. **Accessing Fields and Methods**: Reflection allows you to get and set field values, and invoke methods on objects, even if they are private or protected. This can be achieved through the `Field`, `Method`, and `Constructor` classes.
4. **Modifying Classes**: Although reflection doesn't allow you to modify the structure of classes themselves (like adding new methods or fields), it does let you interact with their existing structure dynamically.

Here's a simple example demonstrating some basic reflection operations:

```
java
Copy code
import java.lang.reflect.Method;
import java.lang.reflect.Field;

public class ReflectionExample {
    public static void main(String[] args) {
        try {
            // Obtain the Class object for the MyClass class
            Class<?> clazz = Class.forName("MyClass");

            // Get and print class name
            System.out.println("Class Name: " + clazz.getName());

            // Create a new instance of MyClass
            Object obj = clazz.getDeclaredConstructor().newInstance();

            // Access and modify a private field
            Field field = clazz.getDeclaredField("privateField");
            field.setAccessible(true); // Bypass private access
            field.set(obj, "New Value");
            System.out.println("Field Value: " + field.get(obj));

            // Invoke a method
            Method method = clazz.getDeclaredMethod("sayHello");
            method.setAccessible(true); // Bypass private access
            method.invoke(obj);

        } catch (Exception e) {
            e.printStackTrace();
        }
```

```
    }
}

class MyClass {
    private String privateField = "Initial Value";

    private void sayHello() {
        System.out.println("Hello, Reflection!");
    }
}
```

Key Points:

- **Performance**: Reflection can be slower than direct method or field access because it involves additional overhead. It also bypasses compile-time type checks, which can lead to runtime errors if not handled carefully.
- **Security**: Using reflection to access private members or bypass security checks can potentially lead to security vulnerabilities. Java provides mechanisms to control access to reflection operations, such as security managers.
- **Use Cases**: Reflection is often used in frameworks and libraries that need to work with classes and objects in a generic way, such as serialization libraries, dependency injection frameworks, and ORM tools. It's also used in testing and debugging scenarios.

While reflection is a powerful tool, it should be used judiciously due to its potential impact on performance and security.

Proxies

In Java, proxies are a mechanism for creating dynamic implementations of interfaces at runtime. Proxies are often used in scenarios where you want to intercept method calls on an object, such as in the implementation of design patterns like the Proxy pattern, or in frameworks and libraries for tasks such as logging, transactions, or security.

Types of Proxies in Java

1. **Dynamic Proxies**: Java provides built-in support for creating dynamic proxies through the `java.lang.reflect.Proxy` class and `InvocationHandler` interface. This allows you to create a proxy instance that implements one or more interfaces and forwards method calls to an `InvocationHandler` instance.
2. **Static Proxies**: Static proxies are manually created by writing a proxy class that implements the same interfaces as the target class and delegates method calls to an instance of the target class. This approach requires more boilerplate code compared to dynamic proxies.

Dynamic Proxy Example

Dynamic proxies are created using the `Proxy` class and an implementation of the `InvocationHandler` interface. Here's a basic example:

```
java
Copy code
import java.lang.reflect.InvocationHandler;
```

```
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

// Define an interface
interface GreetingService {
    void greet(String name);
}

// Implement the interface
class GreetingServiceImpl implements GreetingService {
    @Override
    public void greet(String name) {
        System.out.println("Hello, " + name + "!");
    }
}

// Implement an InvocationHandler
class GreetingInvocationHandler implements InvocationHandler {
    private final Object target;

    public GreetingInvocationHandler(Object target) {
        this.target = target;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        System.out.println("Before method: " + method.getName());
        Object result = method.invoke(target, args);
        System.out.println("After method: " + method.getName());
        return result;
    }
}

public class ProxyExample {
    public static void main(String[] args) {
        // Create the real object
        GreetingService realService = new GreetingServiceImpl();

        // Create the proxy instance
        GreetingService proxy = (GreetingService) Proxy.newProxyInstance(
                GreetingService.class.getClassLoader(),
                new Class[]{GreetingService.class},
                new GreetingInvocationHandler(realService)
        );

        // Use the proxy instance
        proxy.greet("John");
    }
}
```

Explanation

1. **Interface**: `GreetingService` is the interface that the proxy will implement.
2. **Real Object**: `GreetingServiceImpl` is the real implementation of the interface.
3. **Invocation Handler**: `GreetingInvocationHandler` is an implementation of the `InvocationHandler` interface. It defines what happens when a method is invoked on the proxy instance.
4. **Proxy Creation**: `Proxy.newProxyInstance()` creates a proxy instance that implements the `GreetingService` interface and delegates method calls to the `GreetingInvocationHandler`.

Static Proxy Example

A static proxy is a manually created proxy class:

```
java
Copy code
// Define an interface
interface GreetingService {
    void greet(String name);
}

// Implement the interface
class GreetingServiceImpl implements GreetingService {
    @Override
    public void greet(String name) {
        System.out.println("Hello, " + name + "!");
    }
}

// Static Proxy class
class GreetingServiceProxy implements GreetingService {
    private final GreetingService realService;

    public GreetingServiceProxy(GreetingService realService) {
        this.realService = realService;
    }

    @Override
    public void greet(String name) {
        System.out.println("Before method");
        realService.greet(name);
        System.out.println("After method");
    }
}

public class StaticProxyExample {
    public static void main(String[] args) {
        GreetingService realService = new GreetingServiceImpl();
        GreetingService proxy = new GreetingServiceProxy(realService);

        proxy.greet("John");
    }
}
```

Key Points

- **Dynamic Proxy**: Useful when you need proxies for many different interfaces or want to avoid writing boilerplate code. Requires the use of reflection.
- **Static Proxy**: More straightforward but requires manual implementation for each proxy class and interface. Useful when you need a specific proxy class for a known interface.

Proxies are commonly used in various frameworks and libraries, including Java's RMI (Remote Method Invocation), AOP (Aspect-Oriented Programming) frameworks, and various dependency injection frameworks. They provide a flexible way to extend and modify the behavior of objects without altering their code.

In Java, object cloning refers to creating a copy of an existing object. This is often used when you need to duplicate an object while maintaining its state but without having to create a new instance manually. Java provides mechanisms to clone objects through the `Cloneable` interface and the `clone()` method of the `Object` class.

Key Concepts

1. **Cloneable Interface**:
   - The `Cloneable` interface is a marker interface (an interface with no methods) that indicates a class allows its objects to be cloned. If a class does not implement `Cloneable` and you attempt to call its `clone()` method, the method will throw a `CloneNotSupportedException`.
2. **Object.clone() Method**:
   - The `clone()` method is a protected method defined in the `Object` class. To use it, a class must override this method and make it public. The default implementation of `clone()` performs a shallow copy of the object.
3. **Shallow Copy vs. Deep Copy**:
   - **Shallow Copy**: Creates a new object with a new reference, but the fields of the new object are references to the same objects as the fields of the original object. Changes to mutable objects referenced by the original and cloned object will affect both.
   - **Deep Copy**: Creates a new object and recursively copies all objects referenced by the original object. This ensures that the cloned object and the original object do not share any mutable objects.

Example of Object Cloning

Here's an example demonstrating how to implement object cloning using the `Cloneable` interface and the `clone()` method:

```
java
Copy code
class Person implements Cloneable {
    private String name;
    private Address address;

    public Person(String name, Address address) {
        this.name = name;
        this.address = address;
    }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        // Call the superclass's clone method
        Person cloned = (Person) super.clone();
        // Perform deep copy for mutable fields if necessary
        cloned.address = new Address(address.getStreet(), address.getCity());
        return cloned;
    }

    @Override
    public String toString() {
        return "Person{name='" + name + "', address=" + address + "}";
    }
}
```

```
class Address {
    private String street;
    private String city;

    public Address(String street, String city) {
        this.street = street;
        this.city = city;
    }

    public String getStreet() {
        return street;
    }

    public String getCity() {
        return city;
    }

    @Override
    public String toString() {
        return "Address{street='" + street + "', city='" + city + "'}";
    }
}

public class CloneExample {
    public static void main(String[] args) {
        try {
            Address address = new Address("123 Main St", "Anytown");
            Person originalPerson = new Person("John Doe", address);

            // Clone the original person
            Person clonedPerson = (Person) originalPerson.clone();

            // Modify the cloned person's address
            clonedPerson.address = new Address("456 Elm St", "Othertown");

            // Print both persons to demonstrate the clone
            System.out.println("Original Person: " + originalPerson);
            System.out.println("Cloned Person: " + clonedPerson);
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
    }
}
```

Explanation

1. **Implementing Cloneable**:
   - The `Person` class implements the `Cloneable` interface and overrides the `clone()` method to make it public. It uses `super.clone()` to perform a shallow copy and then manually performs a deep copy of the mutable `address` field.
2. **Deep Copy**:
   - In the `clone()` method, a new `Address` object is created for the cloned `Person` to ensure that the cloned `Person` has its own `Address` instance, separate from the original.
3. **Usage**:

- In the `CloneExample` class, the `clone()` method is used to create a duplicate of the `Person` object. Changes to the address of the cloned `Person` do not affect the address of the original `Person`, demonstrating that the deep copy was successful.

Key Points

- **Implementing `Cloneable`**: Classes must implement `Cloneable` to use the `clone()` method without throwing `CloneNotSupportedException`.
- **Overriding `clone()`**: Subclasses need to override `clone()` to define how the cloning is performed, especially if a deep copy is required.
- **Default Behavior**: The default `clone()` method performs a shallow copy. If your class has mutable fields, you may need to implement a deep copy to avoid shared mutable state.

Object cloning can be a powerful feature but should be used with caution, especially when dealing with mutable objects and deep cloning.

User defined Exception
To create your own package, you need to understand that Java uses a file system directory to store them. Just like folders on your computer:

```
Example
└── root
  └── mypack
    └── MyPackageClass.java
```

To create a package, use the package keyword:
MyPackageClass.java

```
package mypack;
class MyPackageClass {
  public static void main(String[] args) {
    System.out.println("This is my package!");
  }
}
```

Save the file as MyPackageClass.java, and compile it:
C:\Users\Your Name>javac MyPackageClass.java

Then compile the package:
C:\Users\Your Name>javac -d . MyPackageClass.java

This forces the compiler to create the "mypack" package.

The -d keyword specifies the destination for where to save the class file. You can use any directory name, like c:/user (windows), or, if you want to keep the package within the same directory, you can use the dot sign ".", like in the example above.

Note: The package name should be written in lower case to avoid conflict with class names.
When we compiled the package in the example above, a new folder was created, called "mypack".

To run the MyPackageClass.java file, write the following:
C:\Users\Your Name>java mypack.MyPackageClass

The output will be:
This is my package!