Sudhanshu BN
1DS18EC091

Introduction to JAVA

1.)

A) d) extends

B) b) class B extends A {}

C) a) True

D) d) all

E) b) Protected

F) c) Throws

G) c) Abstract class can be initiated by new operator

H) c) Byte stream

I) a) IOException

J) a) BufferedReader

2.) Interface

- Interface in JAVA is blueprint of class. It has static constants & abstract methods.
- Interface is a mechanism to achieve abstraction.
- there can be only abstract methods, not body.
- Used to achieve asstraction & multiple inheritance.
- Represents IS-A relationship.
- Interface can't be instantiated.
- The fields can appear in interface must be declared both static & final.
- An Interface can extend multiple interfaces.

Declaring interfaces

interface keyword is used. Interface is an abstract. ~~Data~~ abstract keyword is not required.

eg:-
```
import java.lang.*;
public interface Interface {
    // final, static fields.
    // abstract method delclarations
}
```

eg:-
```
interface Callback {
    void callback(int param);
}
```

Implementation

- One more class can implement interface.
- implement class looks like.

```
class classname [extends superclass] [implements interface [interface ...]]
{ //class body
}
```

Example 1:

```
class client implements Callback {
    // call back interface
    public void callback(int p) {
      System.out.println("call back called with " + p);
    }
    void anotherMethod() {
      System.out.println("classes that implement interfaces" +
                         "may also define other members");
    }
}
```

~~Example 2:~~

When implementing interfaces, there are several rules.

- A class can implement more than one interface at a time.
- A class can extend only one class, but implement many interfaces.
- An interface can extend another interface, in a similar way as a class can extend another class.
- It is both permissible and common for classes that implement interfaces to define additional members of their own.

3.) Exception: An exception is an unwanted event that interrupts the normal flow of the program. When an exception occurs program execution terminates.

i) try block

- A try block is used in exception handling which contains set of statements where an exception can occur
- It is always followed by catch block or finally or both.

syntax:

```
try {
    // statements may cause exception
}
```

ii) Catch block.

- To guard against and handle run time error, simply enclose the code you want to monitor with try block.
- After try block we use catch clause that specifies the type of exception you wish to catch.

syntax:

```
try {
    // Statements may cause exception.
}
catch { // ~~catch~~ type of exception.
}
```

example: try & catch

```
class exc2{
  public static void main (String args[]){
    int d, a;
    try{ // monitor block of code.
      d=0
      a=42/d
      System.out.println("will not be printed");
    }
    catch(ArithmeticException e){ // divide by zero error.
      System.out.println("Division by zero");
    }
```

try block

catch block.

~~throw~~ ii) Throw block.

- Throw is used to throw an exception explicitly.
- It is used for custom exception.

Syntax:

```
throw exceptionObj.
eg: throw new IOException("device error");
```

Example:

```
public class TestThrow1{
  static void validate(int age){
    if(age<18)
      throw new ArithmeticException("not valid");
    else
      System.out.println("You can vote");
  }
```

// For input of age less than 18 it will throw error like "not valid".

iv) Finally block.

- Just as final keyword we have finally which is used after try block.
- It is used with try/catch block & guarantees that this section code will be executed, even if there is an exception.

~~example~~ Syntax:

```
try {
    //
}
catch {
    //
}
finally { // guaranteed execution.
}
```

example:

```
class Dsce {
    static void Student()
    {try System.out.println("Dsce");
        throw new RuntimeException("demo");
    }
    finally
    {System.out.println("Guaranteed execution");
    }
}
```

// finally block will be executed

4.) A) Super keyword

- Super keyword in JAVA is a reference variable which is used to refer immediate parent class object
- when we create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable

Uses

- It can be used to refer immediate parent class instance variable
- It can be used to invoke immediate parent class method.
- It can be used to invoke immediate parent class constructor.

Example to invoke parent class method.

```
class College{
  void study(){ System.out.println("Studying...");}}
  }
class students extends college{
  void study(){ System.out.println("Studying Java...");}}
  void exam{ System.out.println("Exams!");}}
  void work() {
    super.study();
    exam();
  }}
class Test { public static void main(String args[]){
    students s = new students();
    s.work();
  }}
```

Output: Studying...
Exams!

4) B) Method overriding prevention using final.

```
class A {
    int a = 8;
    final void print() {
        System.out.println("Value of a :" +a);
    }
}
class B extends A {
    int b = 3;
    void print() {
        System.out.println("Value of b :" +b);
    }
}
public class Test {
    public static void main (String args[]) {
        B obj = new B();
        obj.print();
    }
}
```

Above program generates error as the method print() in class A is final & so could be overridden by method print() in class B

output:

Test.java: 12: error: print() in B cannot override — print() in A

6) read input from console in Java.

4 types:

1.) Using BufferReader class
2) Using Scanner class
3.) using Console class
4.) Using Cmd line argument.

1.) Using BufferedReader class

Introduced in JDK, used by wrapping System.in in an InputStreamReader which is wrapped in BufferedReader.

→ Input is buffered for efficiency
→ Wrapping code is hard to remember

Example:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
public class Test {
    public static void main (String[] args)
        throws IOException
    { BufferedReader reader = new BufferedReader (new
                              InputStreamReader (System.in));
      //using readLine.
      String name = reader.readLine ();
      System.out.println (name);
    }
}
```

Input: Sudhanshu
Output: Sudhanshu

2) Using Scanner class

Most preferred method to take input - used to parse primitive types & strings.

- Has easier methods for parsing primitives (nextInt(), nextFloat(), ...) from the tokenized input.
- Regular expressions can be used to find tokens
- The reading methods are not synchronized.

Examples:

```
import java.util.Scanner;
class Test {
    public static void main (String args[])
    {
        //using scanner
        Scanner in = new Scanner (System.in);
        String s = in.nextLine();
        System.out.println("You entered string " + s);
        int a = in.nextInt();
        System.out.println("You entered integer " + a);
        float b = in.nextFloat();
        System.out.println("You entered float " + b);
        //close scanner.
        in.close();
    }
}
```

Input: Sudhanshu
3
3.4.

Output: You entered string Sudhanshu
You entered integer 3
You entered float 3.4