MULUND

VASHI

BORIVALI DOMBIVALI DADAR

Phone Nos. 02264205700, 02264402060, 9223376244, 9821882868

NOTES ON

OCA/OCP Java SE 6 & 7

(Exams 1Z0-851 , 1Z0-803 , 1Z0-804)

Book 2

BY

Sandeep J. Gupta

www.study-circle.org

Photo

Receipt No. 187

Date : 28/06/16

SANDEEP GUPTA'S STUDY CIRCLE — CIRCLE OF EXCELLENCE

Service Tax Reg. No. AHHPG7487PST001
Commercial Training & Coaching

Received with thanks from Mr./Ms. Duavesh Tuse

the sum of rupees Seven thousand only

by Cash/Cheque no. 327247 dated 27/6/16 drawn on bank The municipal

for the Subject OCJP Day Sunday Time 3:30-7:00

Sem V Branch COMP College SSJC Residence Mulund

Subject Fees	7000/-	Contact No.1	8452802158
Service Tax	INC	Contact No.2	9869430284
Total	7000/-		

For STUDY CIRCLE

Authorised Signatory

- This receipt should be brought by student in all the lectures.
- Please ask the staff for tentative batch starting date and call us one day before that date to confirm the same .
- Accepting this receipt means you also accept the terms and conditions given overleaf.
- A fine of Rs.50/- will be charged if this receipt is lost.

Mulund (W) Branch : 402, Takshashila Commercial Centre, Near ICICI Bank, R. H. B. Road, Mulund (W), Mumbai - 400 080 Tel. : 02264205700, 9223376244, 98

Vashi Branch : D- 613, 4th Floor, Vashi Plaza, Sector No. 17, Vashi, Navi mumbai. Tel. : 02264402060, 9223376244, 9821882868

www.study-circle.org

Table of Contents

static → Can be accessed using class name

Different types of errors

Errors may broadly be classified into the following two categories:

a. Compile-time errors:

i. Errors detected and displayed by the Java compiler are known as compile-time errors. E.g. syntax errors

ii. Whenever the compiler displays an error, it will not create the .class file.

iii. Hence, it is necessary to fix all the compile-time errors to successfully compile and run the program.

b.Run-time errors:

i. Sometimes, a program may compile successfully creating the .class file but may not run properly. The program may produce wrong results due to wrong logic or may terminate due to certain mistakes in the input.

ii. Such errors which occur while the program is running (and not during compiling) are called run-time errors. Most common run-time errors are:

- Dividing an integer by zero
- Accessing an element that is out of the bounds of an array
- Trying to store a value into an array of an incompatible class or type
- Attempting to use a negative size for an array
- Converting invalid string to a number

iii. When such errors are encountered, Java generates an error message and aborts the program.

Exception

1. An exception is an abnormal condition that arises during the **execution** of a program. In other words, an exception is a run-time error.
2. When an Exception occurs the normal flow of the program is disrupted and the program terminates abnormally.
3. When the JVM encounters an error such as dividing by zero, it throws an exception i.e. it creates an exception object and throws it.
4. Once thrown, the exception object must be caught and handled properly. If this is not done, the JVM will display an error message and abnormally terminate the program. Consider the program shown below:

```
import  java.io.*;
class  NoTryCatch
{
public static void main(String args[]) throws IOException
{
InputStreamReader isr=new InputStreamReader(System.in);
BufferedReader stdin=new BufferedReader(isr);

int a,b,c;

System.out.println("Enter integers a and b: ");
a=Integer.parseInt(stdin.readLine());          b=Integer.parseInt(stdin.readLine());

c=a/b;

System.out.println("a/b = "+c);  } }
```

O/P: Enter integers a and b:
4
0
Exception in thread "main" java.lang.ArithmeticException
/ by zero at NoTryCatch.main
(NoTryCatch.java:17)

Note: In above Eg Program, if a, b, c are double, and b = 0, then
O/P :- a/b = infinity & Exception is not thrown by Exception handler

try catch

1. The basic concepts of exception handling are throwing an exception and catching it.
2. Java uses the keyword **try** to preface a block of code that is likely to cause an error condition and 'throw' an exception.
3. A catch block defined by the keyword **catch** 'catches' the exception 'thrown' by the try block and handles it appropriately. The catch block is added ***immediately*** after the try block.
4. General Format:

```
try
{
   statement-block;          //generates an exception
}
catch(Exception-type e)
{
   statement-block;          //processes the exception
}
```

5. The try block can have one or more statements that could generate an exception. If any one statement generates an exception, the remaining statements in the block are ***permanently skipped*** and execution jumps to the catch block that is placed next to try block.
6. The catch block too can have one or more statements that are necessary to process the exception.
7. Every try statement should be followed by at least one catch statement (or finally), otherwise compilation error will occur.
8. catch statement works like a method definition. The catch statement is passed a single parameter, which is reference to the exception object thrown(by try block). If the catch parameter matches with the type of exception object, then the exception is caught and statements in the catch block will be executed.
9. If the exception is not caught by the catch block, then the default exception handler will cause the execution to terminate.

→ Throwing the object means passing the object as a parameter to the catch block.
→ Catching the object means accepting the object as a parameter and storing it in 'e'.

```
import java.io.*;
class TryCatch
{
public static void main(String args[]) throws IOException
{
InputStreamReader isr=new InputStreamReader(System.in);
BufferedReader stdin=new BufferedReader(isr);
int a,b,c;
System.out.println("Enter integers a and b: ");
a=Integer.parseInt(stdin.readLine());
b=Integer.parseInt(stdin.readLine());

try
{
c=a/b;
}
catch(ArithmeticException e)
{
System.out.println("Wrong Input. b cannot be zero");
System.out.println("Re-Enter b:");
b=Integer.parseInt(stdin.readLine());
c=a/b;
}
System.out.println("a/b = "+c);  }}
```

O/P: Enter Integers a and b:
6
0
Wrong Input. b cannot be zero
java.lang.ArithmeticException : / by zero
Re-Enter b:
3
a/b = 2

① S.O.println(e);
[Same as SOP(e.toString());]
② c=a/b; ← (7th line)

→ If a, b, c is of data type double, the exception will not be thrown if b is 0, rather "Infinity" will be displayed

(Note for Note ④: Whenever we object instead of S.O.P. it acts as an automatic method call to the "toString()" method, hence statement ④ is equivalent to S.O.P(e.toString()). This toString() method is redefined in class ArithmeticException and it returns the string "java.lang.ArithmeticException: / by zero" which is then displayed by S.O.P.

(7) If b is again 0 for statement (2), then statement 2 generates an exception which remains uncalled uncaught.

(3) Here if we write sop(e) in catch block then. O/P will be "java.lang.ArithmeticException: / by zero. The catch block can be kept empty

Multiple Catch

1. In some cases, more than one exception could be raised by a single piece of code.
2. In this type of situation two or more catch clauses can be specified, each handling a different type of exception.
3. Given below is the general format of multiple catch:

```
try
{
  statement-block;                    //generates an exception
}
catch(Exception-type-1   e)
{
  statement-block;                    //processes exception type 1
}
catch(Exception-type-2   e)
{
  statement-block;                    //processes exception type 2
}
.
.
.
catch(Exception-type-N   e)
{
  statement-block;                    //processes exception type N
}
.....
.....
```

4. Java treats multiple catch statements as switch statement. When an exception is thrown, each catch statement is inspected in order, and the first catch whose type matches that of the exception thrown is executed.
5. After one catch statement executes, the others are ***bypassed*** and execution continues after try/catch block.
6. Java actually does not require any processing of the exception at all. We can simply have a catch statement with an empty block to avoid program abortion. This is as shown:

```
catch(Exception e)
{ }
```

The above catch statement does nothing. The statement will catch an exception an ignore it.

Given below is an example:

```
import java.io.*;

class MulCatch
{
public static void main(String args[]) throws IOException
{
InputStreamReader isr=new InputStreamReader(System.in);
BufferedReader stdin=new BufferedReader(isr);

int a,b,c;

try
{
System.out.println("Enter integers a and b: ");
a=Integer.parseInt(stdin.readLine());
b=Integer.parseInt(stdin.readLine());
c=a/b;
}
catch(ArithmeticException e)
{
System.out.println("Wrong Input. b cannot be zero");
System.out.println("Re-Enter a and b:");
a=Integer.parseInt(stdin.readLine());
b=Integer.parseInt(stdin.readLine());
c=a/b;
}
catch(NumberFormatException e)
{
System.out.println("Wrong Input. a and b both should be integers");
System.out.println("Re-Enter integers a and b:");
①a=Integer.parseInt(stdin.readLine());
②b=Integer.parseInt(stdin.readLine());
③c=a/b;
}
System.out.println("a/b = "+c);
}
}
```

O/P 1) Enter integers a and b
4
6.5
Wrong Input. a and b should be integers
Re-Enter integers a and b:
4
2
a/b=2

2) Enter integers a and b:
4 2
Wrong Input. a and b both should be integers.
Re-Enter integers a and b:
4
2
a/b = 2

[every O/P should be on next line, ie we should press Enter after every I/P]

Note :- 1) Here, in try block, there are multiple statements which is valid. All the statements may throw different exceptions which are handled by their respective catch block.

2) While giving 2 inputs, space is not allowed

3) As per O/P 1, statement ② will throw NFE

4) As per O/P 2, statement ① will throw NFE, this is because in java every I/P should be given on a new line.

Q4. Explain how finally statement is implemented in Java.

1. Java supports a statement known as *finally* that can be used to handle an exception that is not caught by any of the previous catch statements.
2. finally block can be used to handle an exception generated within the try block. It may be added immediately after the try block or after the last catch block as shown in the following general format:

```
try
{
  ......
  ......
}
finally
{
 .....
 .....
}
```

```
try
 {
  .....
  .....
 }
catch(...)
{
  ....
  ....
 }
 catch(...)
 {
     .....
     .....
 }
   ..........
   ..........
finally
{
    ........
    .........
}
```

3. Whether an exception is thrown or not, whether an exception is caught or not caught the finally block is guaranteed to execute. Hence we can use it to perform certain operations which are required to be performed at any cost.
4. The finally clause is optional. However, each try statement requires at least one catch or finally clause following the try statement.
5. If there is a return statement in try block, then this return statement will execute only after the finally block executes.

→ Return terminates the function()
→ System.exit(1) terminates the program


```
import java.io.*;
class Finally
{
public static void main(String args[]) throws IOException
{
InputStreamReader isr=new InputStreamReader(System.in);
BufferedReader stdin=new BufferedReader(isr);

int a,b,c;

try
{
System.out.println("Enter integers a and b: ");
a=Integer.parseInt(stdin.readLine());
b=Integer.parseInt(stdin.readLine());
c=a/b;
System.out.println("a/b = "+c);
// System.exit(1);
// return;
}
catch(ArithmeticException e)
{
System.out.println(e);
}
finally
{
// return;  error: unreachable statement
System.out.println("This Statement should always execute");
}
System.out.println("Statement outside finally");
}
}
```

Note :- 1) Statement ① if written prevents execution of finally block and "Run fails"

2) Statement ② if written, finally block will execute immediately after which the program successfully terminates. Note that Statement ④ does not execute.

3) If we write return in finally block then we get compile time error. But we do not get compile time error if we write System.exit(1) in finally block.

O/P: Enter Integers a and b:

4
2
5
This statement should always execute
Statement outside finally

~~Now, Number Format Exce is thrown and program abruptly terminates~~

~~→ What is O/P if~~ → what is O/P if statement ① is uncommented

Enter Integers a and b
8
4
a/b = 2

After this the program terminates abruptly without executing finally block since statement ① will get skipped

If it was 4-1 instead of 4 then the O/P would include finally

Q.5 Explain the concept of nested try.

1. In java, a try-catch statement can be inside the block of another try-catch.
2. If an inner try statement does not have a matching catch statement for a particular exception, then the next try statement's catch handlers are checked for a match. This continues until one of the catch statements succeeds, or until all of the nested try statements are done .
   If none of the catch statements match, then the Java run-time system will handle the exception and will report an error.
3. In other words, an inner try can use catch block of an outer try(if the exception matches). However, the vice-versa is not allowed.
4. Consider the following examples:

```
class  NestedTryCatch
{
public static void main(String args[])
{
int n;
try
{
 try
 {
 n = Integer.parseInt(args[0]);
 System.out.println(5/n);
 }
 catch(ArithmeticException e)
 { System.out.println("Inner Catch"); }
}
 catch(NumberFormatException e)
 { System.out.println("Outer Catch"); }
}
}
```

O/P: -

```
Java Nested Try Catch
0
Inner Catch
```

```
Java Nested Try Catch 6.4
Outer Catch
```

```
Java Nested Try Catch 1 4 6
Outer Catch
```

```
Java Nested Try Catch 1 0
[Now program abruptly terminates and
A Exception occurs]
```

Note :-

1) The second O/P shows that the inner try can use the outer catch if the exception types are matching

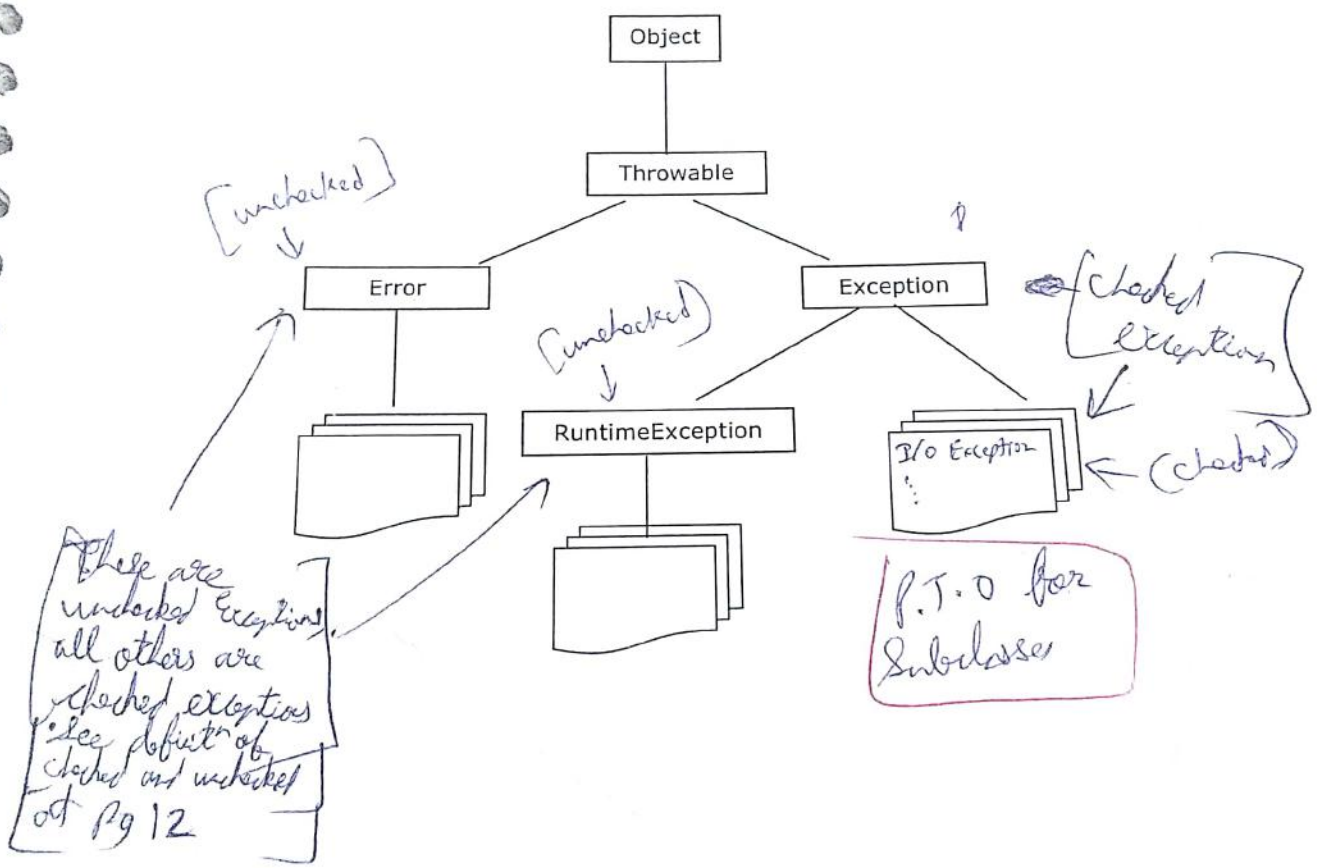
2) The 4th O/P shows that the outer try cannot use the inner catch even if the exception types are matching. This is because the inner catch gets permanently skipped if exception arises (as all statements are skipped in try, when exception arises)

```
class  NestedTryCatch1
{
public static void main(String args[])
{
int n;
try
{
n = Integer.parseInt(args[0]);
System.out.println(5/n);
try
{ }
catch(ArithmeticException e)
{ System.out.println("Inner Catch"); }
}
catch(NumberFormatException e)
{ System.out.println("Outer Catch"); }
}
}
```

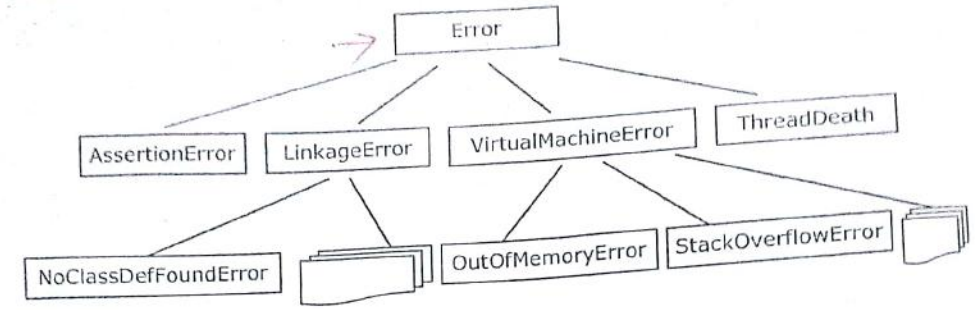
→ You cannot catch subclass of class Error and Java will handle itself

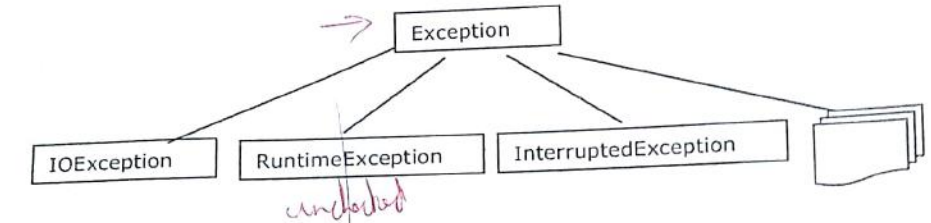
Q6. Explain Exception types or Exception hierarchy in java.

1. All Exception types in java are subclasses of a pre-defined class called ***Throwable***. In other words, Throwable is the superclass of all those classes which handle exceptions in java. Throwable itself is the subclass of ***Object***.
2. ***Throwable*** has two important subclasses : ***Exception*** and ***Error***.
3. ***Exception*** class is used for those exceptional conditions that user programs should catch. It is also the class which we will normally subclass in our programs to create and throw user-defined exceptions. Exception itself has many built-in subclasses like RuntimeException, IOException, InterruptedException etc. Of all these classes we normally use subclasses of RuntimeException.
4. SubClasses of class ***Error*** represent unusual situations that are not caused by program errors. The Error class of java defines those errors which are not expected to be caught by our programs. This class is normally used by the java run-time system itself to handle some very serious problems like JVM running out of memory, Stack Overflow Error etc.

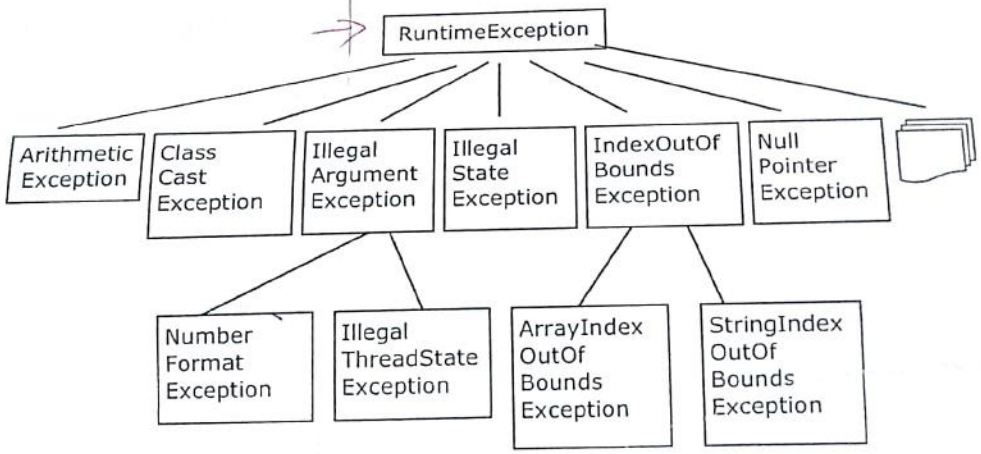


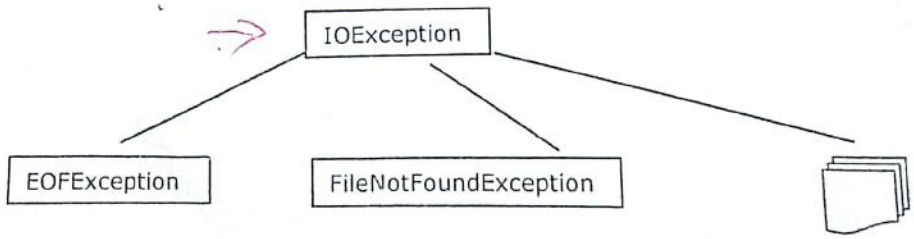
Learn the hierarchy before exam





unchecked





usually
Subclasses of Error are not handled by program, but by JVM
but we can catch subclasses of Error.

Exception Type	Description
ArithmeticException	Thrown by Arithmetic Errors, such as divide by zero
ArrayIndexOutOfBounds Exception	Thrown when Array index is out of specified boundary (either negative or beyond the length of array)
ArrayStoreException	Thrown while Assigning a value of incompatible type to an array element
ClassCastException	Thrown when attempting to cast a reference variable to a type that fails the IS-A test.
IllegalArgumentException	Thrown when a method receives an argument formatted differently than the method expects.
IllegalStateException	Thrown when the state of the environment doesn't match the operation being attempted – for example, using a scanner that's been closed.
NegativeArraySizeException	Thrown when an array is created with negative size
NullPointerException	Thrown when attempting to invoke a method on a reference variable whose current value is null.
NumberFormatException	Thrown when a method that converts a string to a number receives a string that it cannot convert.
StringIndexOutOfBounds Exception	Thrown when string index is out of specified boundary (either negative or beyond the length of string)
AssertionError	Thrown when an assert statement's boolean test returns false.
ExceptionInInitializer Error	Thrown when attempting to initialize a static variable or an initialization block.
NoClassDefFoundError	Thrown when the JVM cant find a class it needs, because of command line error, a classpath issue or a missing .class file.
StackOverflowError	Thrown when a method recurses almost infinitely.

Q.7. Explain the throws keyword OR
Explain the checked and unchecked exceptions in java.

1. In java exceptions can be classified as:
   i. Unchecked Exceptions
   ii. Checked Exceptions

2. **Unchecked Exceptions:**
   a. In the package java.lang, several exception classes such as ArithmeticException, NumberFormatException, ArrayIndexOutOfBoundsException, NegativeArraySizeException etc. are defined. These exceptions are subclasses of RuntimeException.
   b. These are known as unchecked exceptions because the **COMPILER *does not check*** if a method handles or throws these exceptions. Therefore handling these exceptions is not compulsory and they need not be declared in any methods' **throws** list.

3. **Checked Exceptions:**
   a. Checked exceptions are those exceptions which must be compulsorily handled.
   b. If a method is capable of causing a checked exception that it does not handle, then the method must declare this exception by including a **throws** clause in the method's declaration.
   c. Declaring a checked exception (if it is not handled) in the method is necessary so that the callers of this method can guard themselves against this exception.
   d. The ***COMPILER reports an error*** if a checked exception is neither handled nor declared.
   e. Examples of checked exception are: IOException, ClassNotFoundException, InterruptedException, NoSuchFieldException, NoSuchMethodException etc.
   f. The general format of throws clause is

```
returntype methodname(argument list) throws exception-list
{
    // body of method
}
```

Here, exception-list is a comma separated list of exceptions that a method can throw
For example,

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

All exceptions EXCEPT subclasses of Error and RuntimeException are checked exceptions.

Eg:

```
public void getRoll() throws IOException
{
  S.O.Pln("Enter roll-number:");
  try
  { roll = Integer.ParseInt(stdin.readLine()); }
  Catch (IOException e)
  { }
}
```

(Throws IO exception called as checked exception)

CORRECT – If try and like both present
COMPILE TIME ERROR – If try and like both absent

Ducking or Passing the Exception:

Example 1:

```
import java.io.* ;
class  Ducking
{
int  a, b ;
public static void main(String args[]) throws IOException
{
      new Ducking().sum();
}
public void sum() throws IOException
{
      readab();
      System.out.println(a+b);
}
public void readab() throws IOException
{
      InputStreamReader isr=new InputStreamReader(System.in);
      BufferedReader stdin=new BufferedReader(isr);

      System.out.println("Enter 2 integers: ");
      a=Integer.parseInt(stdin.readLine());
      b=Integer.parseInt(stdin.readLine());
}
}
```

Example 2:

```
import java.io.* ;
class  Ducking1
{
int a, b ;
public static void main(String args[])
{
      try
      {
      new Ducking().sum();
      }
      catch (IOException  e)    { System.out.println(e); }
}
public void sum() throws IOException
{
      readab();
      System.out.println(a+b);
}
public void readab() throws IOException
{
      InputStreamReader isr=new InputStreamReader(System.in);
      BufferedReader stdin=new BufferedReader(isr);

      System.out.println("Enter 2 integers: ");
      a=Integer.parseInt(stdin.readLine());
      b=Integer.parseInt(stdin.readLine());
}
}
```

Q.8 Explain mechanism of creating & throwing our own exception.

Sometimes it may be necessary to throw our own user defined exceptions to handle situations specific to our application. For this, we have to use the keyword 'throw'.

Program to read an integer & throw an exception if the integer is divisible by 7

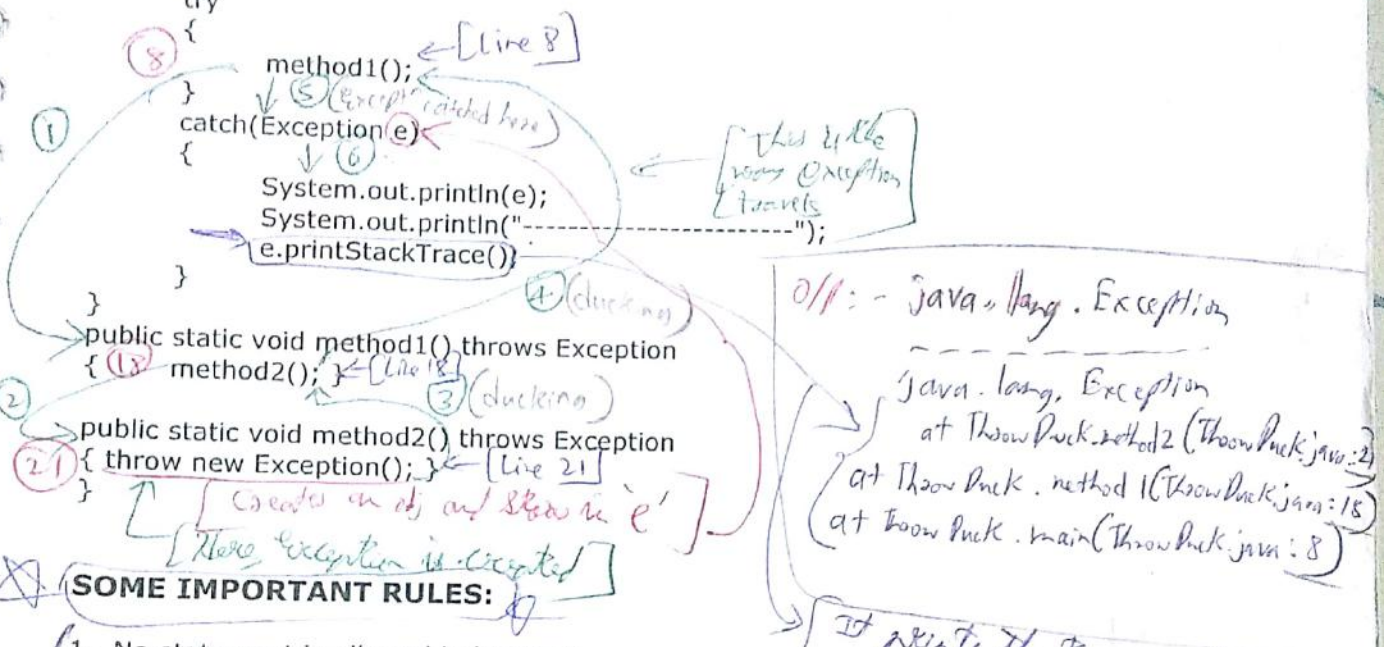
```
import java.io.*;
class Throw
{
public static void main(String args[]) throws IOException
{
InputStreamReader isr=new InputStreamReader(System.in);
BufferedReader stdin=new BufferedReader(isr);

int n;

try
{
System.out.println("Enter an integer: ");
n=Integer.parseInt(stdin.readLine());
if(n % 7 == 0)
throw new Exception("Given Integer is divisible by 7");
System.out.println("Given Integer is "+n);
}
catch(Exception e)
{
System.out.println(e);
}
System.out.println("Program Over");
}
}
```

Example:

```
import java.io.*;
class Throw1
{
public static void main(String args[]) throws IOException // Statement 2
{
InputStreamReader isr=new InputStreamReader(System.in);
BufferedReader stdin=new BufferedReader(isr);

int n;

System.out.println("Enter an integer: ");
n=Integer.parseInt(stdin.readLine());
if(n % 7 == 0)
throw new Exception("Given Integer is divisible by 7"); // Statement 1
System.out.println("Given Integer is "+n);

System.out.println("Program Over");
}
}
```

Method Stack:

```
import java.io.* ;
class ThrowDuck
{
public static void main(String args[]) throws Exception
{
    try
    {
        method1();
    }
    catch(Exception e)
    {
        System.out.println(e);
        System.out.println("----------");
        e.printStackTrace();
    }
}
public static void method1() throws Exception
{   method2(); }
public static void method2() throws Exception
{ throw new Exception(); }
```



O/P :- java.lang.Exception

java.lang.Exception
at ThrowDuck.method2(ThrowDuck.java:2)
at ThrowDuck.method1(ThrowDuck.java:18)
at ThrowDuck.main(ThrowDuck.java:8)

[It prints the traces of stack of ducking]

SOME IMPORTANT RULES:

1. No statement is allowed in between
   a. try block and catch block
   b. catch block and finally block
   c. try block and finally block.
2. finally block cannot come before any catch block
3. finally block can be given only once.
4. finally block can be written without catch but not without try.
5. Following is an invalid example of multiple catch:

```
class MulCatchTest
{
public static void main(String args[])
{
try
{ System.out.println(6/0); }
catch(RuntimeException e)
{ System.out.println("AE"); }
catch(ArithmeticException e)
{ System.out.println("RE"); }
}
}
```

V.Imp

Note
AE is subclass of RuntimeException, so AE should be written before RE, bcoz if we write RE before, AE will not be executed everytime (ie) Superclass cannot come above subclass

Rules regarding Method Overriding

1. The overriding method can throw **ANY unchecked** exception regardless of whether the overridden method declares that exception.
2. The overridden method must **NOT** throw any checked exception that is **NEW** or **BROADER** than the one declared by the overridden method. In other words, the overridden method can throw **FEWER** exceptions or can throw exceptions **NARROWER** than the one declared by the overridden method.

```
import java.io.*;
class SuperClass  {
      void show() throws ArrayIndexOutOfBoundsException   (1)
      { }

      void display() throws IOException
      { }
}

class SubClass extends SuperClass {
      void show() throws ArithmeticException   (2)
      { }

      void display() throws FileNotFoundException   // , InterruptedException
      { }                                            (3)
}
```

Not. 1> If we interchange statement (1) and (2) the compiler will report an error bcoz, overriding method exception is broader than overridden method.

2> If we write ":- ", Interrupted Exception " after statement (3) compiler will report an error, bcoz ~~this exception~~ bcoz, it is new exception.

Rule regarding Method Overloading

Overloaded methods can declare new or broader checked exceptions. Basically, there is **NO RULE** for overloaded methods.

```
import java.io.*;
class A
{
      void show() throws ArrayIndexOutOfBoundsException
      { }

      void show(int x) throws FileNotFoundException
      { }
}
```

Q. Explain Assertions.

- Assertion is a statement in java. It can be used to test our assumptions about the program.
- While executing assertion, it is believed to be true. If it fails, JVM will throw an error named AssertionError. It is mainly used for testing purpose.
- Assertions was Added to the java language beginning with version 1.4.
- Prior to java 1.4 assert could be used as an identifier.

Flavors of Assertion:

- Assertions come in two flavors: **Really Simple and Simple**

Example	
• Really Simple:	assert n > 0 ;
• Simple:	assert n > 0 : "Given integer is not positive" ;

- The difference between the two is that the simple version adds a second expression, separated from the first expression (boolean expression) by a colon, this expression's string value is added to the stack trace.
- Both versions throw an immediate AssertionError, if the first expression is false. The simple version gives us a little more debugging help, while the really simple version simply tells us that our assumption was false.

Assertion Expression Rules [Self Study]

- Assertions can have either one or two expressions, depending on whether you're using the "simple" or the "really simple" version.
- The first expression must always result in a boolean value! Follow the same rules you use for if and while tests.
- The second expression, used only with the simple version of an assert statement, can be anything that results in a ***value***.
- The following code lists legal and illegal expressions for both parts of an assert statement.

```
void noReturn () { }
int  aReturn ()  { return 1; }
void go () {
  int x = 1;
  boolean b = true;
  // the following six are LEGAL assert statements
  assert ( x == 1) ;
  assert (b) ;
  assert true;
  assert ( x == 1) : x;
  assert ( x == 1) : aReturn() ;
  assert ( x == 1 ) : new Box() ; // Box is some class
 // the following six are ILLEGAL assert statements
 assert (x = 1) ;       // none among first 3 have boolean expression
 assert (x) ;
 assert 0;
 assert (x == 1) : ;                  // none among last 3 return a value
 assert (x == 1) : noReturn() ;
 assert (x == 1) : Box b1 ;          // b1 is reference variable, not object
```

Enabling and Disabling of Assertion :

- Enabling assertions means they will be execute at runtime.
- By default , assertions are disabled. Their execution is then effectively equivalent to empty statement .
- Typically, aasertions are enabled during development (or testing) and left disabled once the program is deployed. (Deployed means now the program is being used in the real world.)
- Since assertions are already in the compiled code, they can be turned on whenever needed.
- The option ***–enableassertions*** , or its short form ***–ea*** , enables assertions, and the option ***–disableassertions*** , or its short form ***–da***, disables assertions.

How Assertion Works :

- Assertions work quite simply. We always assert that something is true. If it is actually true, no problem. Our code keeps running.
- But if our assertion turns out to be wrong (false), then an AssertionError is thrown which we should never, ever handle.
- That means we shouldn't catch it with a catch clause and attempt to recover.

Examples: Program to read an integer and display the same if it is positive.

1) Really Simple

```
class  Assert1
{
public static void main(String args[])
{
int  n ;
n=Integer.parseInt(args[0]);
assert   n > 0 ;                    // space required after assert
System.out.println("n = " + n);
}
}
```

o/p :- C:\oajp > java Assert1 6
n = 6
... java Assert1 -6
n = -6
java -ea Assert1 -6
Exception in thread "main" java.lang.AssertionEr...
java -ea Assert1 6
n = 6

By default assertion is disabled
If we want to enable it then:
∴ -ea ⇒ enable assertion.

ea is a flag and '-' should be put before it

2) Simple

```
class  Assert2
{
public static void main(String args[])
{
int  n ;
n=Integer.parseInt(args[0]);
assert(n > 0) : "Given integer is not positive" ;  //No space but parentheses
System.out.println("n = " + n);
}
}
```

o/p : java -ea Assert2 6
n = 6
java -ea Assert2 -6
Exception in thread "main" java.lang.
AssertionError : Given integer is not positive.

The AssertionError Class

- The java.lang.AssertionError class is a subclass of java.lang.Error.
- Thus AssertionErrors are unchecked.
- They can be caught and handled using the try-catch construct, and the execution continues normally, as one would expect.
- However, Errors are seldom caught and handled by the program, and the same applies to AssertionErrors.

What is Appropriate and What is not ?

1) It is appropriate to use assertions to validate Arguments to a private Method.
2) It is not appropriate to use assertions to validate Arguments to a public Method.
3) It is not appropriate to use assertions to validate Command-Line Arguments.
4) It is not appropriate to catch and handle assertions.
5) It is not appropriate to use assertion expressions that can cause side Effects.
6) It is not appropriate for assertions to change a program's state.
7) It is appropriate to use assertions to generate alerts when you reach code that should not be reachable.

Compiling Assertion-Aware code:

- All compilers from Java 4 onwards consider assert as a keyword by default. These compilers will generate an error message if they find the word *assert* used as an identifier.
- However, you can tell the compiler that you're giving it an old piece of code to compile and that it should pretend to be an old compiler! Let's say, you've got to make a quick fix to an old piece of 1.3 code that uses assert as an identifier.
- At the command line, you can type : *javac -source 1.3 Oldcode.java*
- The compiler will **issue warnings** when it discovers the word assert used as an identifier, but the code will compile and execute. Suppose you tell the compiler that your code is version 1.4 or later ; for instance: *javac -source 1.4 Filename.java*
- In this case, the compiler will issue errors when it discovers the word assert used as identifier.
- If you want to use assert as an identifier in your code, you **MUST** compile using the -source 1.3 option.
- The following table summarizes how the java 7 compiler will react to assert as either an identifier or a keyword.

Command Line	If assert is an Identifier	If assert is a Keyword
javac -source 1.3 TestAsserts.java	Code compiles with warnings	Compilation fails
javac -source 1.4 TestAsserts.java	Compilation fails	Code compiles
javac -source 1.5 TestAsserts.java javac -source 5 TestAsserts.java	Compilation fails	Code compiles
javac -source 1.6 TestAsserts.java javac -source 6 TestAsserts.java	Compilation fails	Code compiles
javac -source 1.7 TestAsserts.java javac -source 7 TestAsserts.java	Compilation fails	Code compiles
javac TestAsserts.java	Compilation fails	Code compiles

Package – Definition & Advantages

1. A package is a collection of classes, interfaces and/or other packages.
2. Advantages :

Code Reusability:
Code reusability is one of the main features of any Object Oriented language. Code reusability in Java can be achieved by inheritance and interfaces. However this is limited to reusing classes within a program. In Java it is possible to use classes from other programs using **packages.** Once a class is in a particular program it can then be used by any program.

Organization:
Packages allow organization of classes into smaller units and make it easier to locate and use the appropriate class file.

Avoiding naming conflicts:
While working with number of classes it becomes difficult to decide on names of classes and methods. One may want to use the same name which belongs to another class. Packages help in avoiding naming conflicts as two classes in two different packages can have the same name.

Package Creation

a) Packages can be created by simply including a package command as the first statement in a java source file. Any class now declared within that file will belong to the specified package.
b) In case the package name is omitted, the classes are put into a default package with no name.
c) General Format of package command is: **package** package-name;
d) There are two ways of working with packages:

Method 1:

❖ **Step 1:** Create a file A.java as shown below in the current folder.

```
package package1;
public class A
{
public void show()
{
S.O.P("class A");
}
}
```

Now, give set path = " address "

❖ **Step 2:** Create a sub-folder called "package1" in the current folder and then move A.java in this sub-folder.

❖ **Step 3:** Using command prompt go to current folder and give the command
javac package1\A.java
This command creates a file A.class in the same package(folder) i.e. package1

Method 2: [We will use This method Always]

- **Step 1:** Same as Above (Space)
- **Step 2:** Using command prompt go to current folder and give the command
  javac -d . A.java → It creates a folder package1 → A.java
  This command first ***creates*** a sub-folder called "package1" and then stores the file A.class in sub-folder package1.
- **Step 3:** Finally move A.java in this sub-folder "package1".

Using any one of the above two steps we create a package called package1 with A.java and A.class. This package is now ready to be used.

Using a Package

There are two ways in which a package can be used:

A) Through package name:
In the current folder (of which package1 is the sub-folder) create the following file as ImportOneClass1.java

```
class ImportOneClass1 →
{
  psvm(String args[])
  {
    Package1.A a1 = new Package1.A();
    a1.show();
  }
}
/* Class A
*/
```

[This class is using class A which is in package Package1, but it is itself not inside package]

Compile Command : javac ImportOneClass1.java
Run Command : java ImportOneClass1

B) Using import Command:
In the current folder (of which package1 is the sub-folder) create the following file as ImportOneClass2.java

```
import Package1.A; →
class ImportOneClass2
{
  psvm(String args[])
  {
    A a1 = new A();
    a1.show();
  }
}
/* class A
*/
```

[Here ~~Package1~~ is class A of package 1 is imported]

Compile Command : javac ImportOneClass2.java
Run Command : java ImportOneClass2

1. The import statement is used to bring certain classes or entire packages into visibility (that is, into a particular program). Once a class has been imported it can be referred **directly using its name**.
2. If many classes are imported in an application it will save a lot of typing.
3. Import statements should be placed immediately after the package statement and before any class definitions.
4. General Format:
   import pkg1[.pkg2].(classname / *);

   Eg: import package1.package2.A;
   import package1.A;
   import package1.*;
   import Dagdu.Laalu.Bhavdu.*;

   pkg1 is the name of a top level package,
   pkg2 is the name of a subordinate package inside the outer package(pkg1) separated by a dot. There is no limit to the depth of package hierarchy.
   It is possible to specify either an explicit classname to be imported or an asterisk(*), which indicates that all the classes within a package is to be imported.
5. For example, **import java.util.Vector;** imports the Vector class from util package which is within the java package. **import java.io.*;** imports all the classes from the io package which is within the java package.

Importing all classes of a package

Lets say we already have package1 with A.java and A.class in it.

- **Step 1:** Create a file B.java as shown below in the current folder.

```
package package1 ;
public class B
{
public void show()
{
 System.out.println("Class B");
}}
```

- **Step 2:** Using command prompt go to current folder and give the command
  javac -d . B.java
  This command stores the file B.class in **already created** sub-folder package1.

- **Step 3:** Finally move B.java in this sub-folder "package1".

- **Step 4:**
  In the current folder (of which package1 is the sub-folder) create the following file as ImportAll.java

```
import package1.*;
class ImportAll
{
public static void main(String args[])
{
      A a1 = new A();      a1.show();
      B b1 = new B();      b1.show();
}}
```

O/P:- Class A
Class B

Compile Command : javac ImportAll.java Run Command : java ImportAll

Package Hierarchy

Package hierarchy means one package inside another package.

Lets say we already have package1 with classes A and B in it.

- **Step 1:** Create a file C.java as shown below in the current folder.

```
package package1.package11 ;
public class C
{
public void show()
{
System.out.println("Class C");
}
}
```

- **Step 2:** Using command prompt go to current folder and give the command
  javac -d . C.java
  This command first creates a sub-folder called "package11" inside "package1" and then stores the file C.class in sub-folder package11.

- **Step 3:** Finally move C.java in this sub-folder "package11".

- **Step 4:**
  In the current folder (of which package**1** is the sub-folder) create the following file as ImportSubpackage

```
import package1.*;
import package1.package11.*;
class ImportSubpackage
{
public static void main(String args[])
{
      A a1 = new A();
      a1.show();

      B b1 = new B();
      b1.show();

      C c1 = new C();
      c1.show();

}
}
```

Compile Command : javac ImportSubpackage.java
Run Command : java ImportSubpackage

Note:
The statement import package1.*; will simply import class A and B but not C.

Static imports

New Topic

1. Consider the following situation in which we have class D in package1 and class ImportStatic1 in current folder.

```
package package1 ;
public class D
{
public static int x = 24 ;
public static void show()
{
 System.out.println("Class D");
}
}
```

```
import package1.D ;
class ImportStatic1
{
public static void main(String args[])
{
      System.out.println("x = "+D.x);
      D.show();
}
}
```

o/p :- x = 24
class D

2. The advantage of import statement is that it
   a. reduces typing effort &
   b. makes code easier to read.
3. From Java 5 onwards, the import statement was enhanced to provide greater reduction in typing effort.
4. This new feature is called static imports and it can be used to access static members of a class.
5. Example:

import all static members of class D

Static cannot come before import

```
import static package1.D.* ;
class ImportStatic2
{
public static void main(String args[])
{
   System.out.println("x = "+x);  // statement 1
   show();                        // statement 2
}
}
```

6. The import statement in this example is called "static import". It should be observed that because of this import statement, we are **not** using classname D to access members x and show() in statements 1 and 2.
7. In this program, compiler reports error if we write D.x or D.show(). Hence after doing static import "**D.**" is not allowed.
8. Moreover, instead of a single import statement we may write two import statements as shown:

```
import static package1.D.x ;
import static package1.D.show ;
```


Normal import, imports classes of package
Static import, imports static members of class of package

General Rules while creating any source file

- There can be *only one public* class per source code file
- Comments can appear at the beginning or end of any line in the source code file; they are independent of any of the positioning rules discussed here.
- If there is a *public* class in a file, the name of the file must match the name of the *public* class. For example, a class declared as *public class Dog { }* must be in a source file named *Dog.java*.
- If the class is part of a package, the *package* statement must be the first line in the source code of the file, before any *import* statements that may be present.
- If there are *import* statements, they must go *between* the *package* statement (if there is one) and the class declaration. If there isn't *package* statement, then the *import* statements(s) must be the first line(s) in the source code file.
  If there are no *package* or *import* statements, the class declaration must be the first line in the source code file.
- *import* and *package* statements apply to all classes within a source code file. In other words, there's no way to declare multiple classes in a file and have them in different packages or use different imports
- A file can have more than one non-public class
- Files with no *public* classes can have a name that does not match any of the classes in the file.

Name conflicts while importing packages

```
package1
package package1;
public class A
{
public void show()
{System.out.println("Package 1");}
}

package2
package package2;
public class A
{
public void show()
{ System.out.println("Package 2");}
}
Package 1 and 2 may have many more
classes
```

```
import package1.*;
import package2.*;

class ImportErrorName
{
public static void main(String args[])
{
// A a1 = new A(); -- 1 -- ERROR
package1.A a1 = new package1.A();
a1.show();
}
}

Output:
Package1
```

Note: package1 and package2 each has a class with name A. Both the classes are imported in the program ImportErrorName. Statement No. 1 would report an error since the compiler does not understand whether to create an object of class imported from package1 or package2. To resolve this name conflict, we need to specify the package name as done in the next statement.

Hiding a Class

1. When a package is imported using asterisk(*), all the public classes are imported. However we may want to avoid import of certain classes.
2. That is, in some cases we may like to hide certain classes from being accessible outside the package. Such classes **should not** be declared public.
3. Consider the following example:

```
package package3;
public class A
{
public void show()
{System.out.println("Class A");}
}

package package3;
class B
{
public void show()
{System.out.println("Class B");}
}
```

```
import package3.*;
class ImportErrorPublic
{
public static void main(String args[])
{
 A a1 = new A();
 a1.show();
 // B b1 = new B();  Error
}
}
```

The compiler will report an error because class B has not been declared public. Hence it is not imported and therefore unavailable for creating objects in class ImportErrorPublic.

Although class B cannot be imported (cannot be used outside its package), it can still be used by any class in its **own package.**

Access Specifiers / Visibility Specifiers

1. Java has four levels of access protection – private, default or friendly, protected and public. These five levels of access protection for class members can be summarized as follows:

2.

Specifier / Accessing Class	private (Same class)	Default or friendly (Same Package)	protected (Same Package + Diff Package Subclass)	public [All]
Same Class	YES	YES	YES	YES
Same Package Subclass	NO	YES	YES	YES
Same Package Non-Subclass	NO	YES	YES	YES
Different Package Subclass	NO	NO	YES	YES
Different Package Non-Subclass	NO	NO	NO	YES

2. Members declared private cannot be accessed outside their class.
3. Members declared default, that is, with no explicit access specifier can be accessed by their own class and by other classes in the same package.
4. Members declared protected can be accessed by their own class, by other classes in the same package and only by a subclass in a different package.
5. Members declared public can be accessed from anywhere.

Same class

Package 1

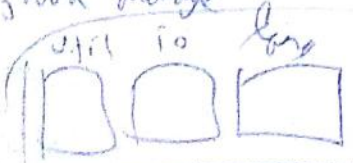
```
class A
{
    int x;
}
class B extends A
{ ...
}
class C
{ ...
}
```

Package 2

```
class X extends A
{ ...
}
class Y
{ ...
}
```

- java.lang package
- has many classes like
  - String
  - StringBuffer
  - Integer
  - Exception
  - Math
  - Wrapper class
  - Object

java package → Classes of java.lang package are implicitly imported.



→ java package java has many subpackage like util, io, lang, etc.

java.util → Scanner, Vector

java.io → ISR, BR

→ A string literal must be opened and closed " " on the same line, else

JAR

- Once you've built and tested your application, you might want to bundle it up so that it's easy to distribute and easy for other people to install.
- One mechanism that java provides for these purposes is a JAR file. JAR stands for Java **AR**chive.
- JAR files are used to compress and archive data (similar to ZIP files).
- Once a JAR file is created, it can be moved from place to place and from machine to machine, and all of the classes in the JAR can be accessed, via classpaths, by java and javac,without ever unJARing the JAR file

Creating a JAR File :

- The basic format of the command for creating a JAR file is :
  - **jar cf jar-file-name input-file-name(s)**
- The options and arguments used in this command are :
  c: indicates that you want to ***create*** a JAR file.
  f: indicates that you want the output to go to a ***file*** rather than to stdout(monitor).
- jar-file-name is the name that you want the resulting JAR file to have
- You can use any filename for a JAR file. By convention JAR filenames are given .jar extension, though this is not required.
- The input-file-name(s) argument is a space-separated list of one or more files that you want to include in your JAR file. The input-file-names(s) argument can contain the wildcard * symbol. If any of the "input-files" are directories, the contents of those directories are added to the JAR archive recursively.
- The c and f options can appear in ***any order***, but there must not be any space between them.
- This command will generate a compressed JAR file and place it in the current directory.The command will also generate a default manifest file for the JAR archive.

Viewing the Contents of a JAR File:

- The format of command for viewing the contents of JAR file is: **jar tf jar-file-name**
- The t option indicates that you want to view the table of contents of the JAR file.
- The f option indicates that the JAR file whose contents are to be viewed is specified on the command line.
- The jar-file-name contains the path and name of the JAR file whose contents you want to view.
- The t and f options can appear in either order, but there must not be any space between them.
- This command will display the JAR file's table of contents to stdout(monitor).

Example 1:
Simulate a scenario in which there is a:

- JAR file j1.jar which contains a
- package called jarpackage1 which contains a
- class Square which contains a
- static method sq which finds square of an integer.

Solution:

❖ **Step 1:** Create a file Square.java as shown below in the current folder.

```
package jarpackage1 ;
public class Square
{
public static int sq(int x)
{
 return x*x ;
}
}
```

❖ **Step 2:** Using command prompt go to current folder and give the command

javac -d . Square.java

This command first ***creates*** a sub-folder called "jarpackage1" and then stores the file Square.class in sub-folder jarpackage1.

❖ **Step 3:** Move Square.java in this sub-folder "jarpackage1".

❖ **Step 4:** Create JAR file by giving command: **jar -cfv j1.jar jarpackage1**

```
added manifest
adding: jarpackage1/(in = 0) (out = 0) (stored 0%)
adding: jarpackage1/Square.class(in = 252) (out = 196) (deflated 22%)
adding: jarpackage1/ Square.java(in = 99) (out = 79) (deflated 20%)
```

❖ **Step 5:** In the current folder (of which jarpackage1 is the sub-folder) create the file as FindSquare.java

```
import jarpackage1.*;
class FindSquare
{
public static void main(String args[])
{
 int s = Square.sq(2) ;
 System.out.println(s) ;
}
}
```

Compile Command : **javac -cp j1.jar FindSquare.java**

Instead of writing "**cp**" we may write "**classpath**" also.

Run Command : **java -cp j1.jar;. FindSquare**

The **;.** should be given immediately after j1.jar **without** any space.

Example 2:
Simulate a scenario in which there is a:
- JAR file j2.jar which contains a
- package called jarpackage1 which contains a
- class Square which contains a static method sq which finds square of an integer.

Add to jarpackage1
- a sub-package jarpackage2 which contains a
- class Max which contains a
- static method maximum which finds maximum of two given integers.

Solution:

❖ **Step 1:** Create a file Max.java as shown below in the current folder.

```
package jarpackage1.jarpackage2 ;
public class Max
{
public static int maximum(int x,int y)
{
if(x>y)
 return x;
else
 return y;
} }
```

❖ **Step 2:** Using command prompt go to current folder and give the command
javac -d . Max.java
This command first creates a sub-folder called "jarpackage2" inside "jarpackage1" and then stores the file Max.class in jarpackage2.

❖ **Step 3:** Move Max.java in the sub-folder "jarpackage2".

❖ **Step 4:** Create JAR file by giving command: **jar -cfv j2.jar jarpackage1**

```
added manifest
adding: jarpackage1/(in = 0) (out = 0) (stored 0%)
adding: jarpackage1/jarpackage2/(in = 0) (out = 0) (stored 0%)
adding: jarpackage1/jarpackage2/Max.class(in = 302) (out = 236) (deflated 21%)
adding: jarpackage1/jarpackage2/Max.java(in = 144) (out = 102) (deflated 29%)
adding: jarpackage1/Square.class(in = 252) (out = 196) (deflated 22%)
adding: jarpackage1/ Square.java(in = 99) (out = 79) (deflated 20%)
```

❖ **Step 5:** In the current folder (of which jarpackage1 is the sub-folder) create the file as FindMaxSquare.java

Import class Square,

```
import jarpackage1.*;
import jarpackage1.jarpackage2.*;
class FindMaxSquare
{
public static void main(String args[])
{
 int m = Max.maximum(2,6);
 int s = Square.sq(m) ;
 System.out.println(s) ;
}}
```

imports class Max

O/P:- 36

Compile Command : **javac -cp j2.jar FindMaxSquare.java**
Run Command : **java -cp j2.jar;. FindMaxSquare**

Three advantage of jar
1) Confidential
2) dealing with only one .jar file.
3) We may compile classes and packages inside jar, without unjarring it.

Using .../jre/lib/ext with JAR files

- When you install java, you end up with a huge directory tree of java related stuff, including the JAR files that contain the classes that come standard with J2SE.
- java and javac have a list of places that they access when searching for class files.
- Buried deep inside your java directory tree is a subdirectory named **jre/lib/ext.**
- If you put JAR files into the ext subdirectory, java and javac can find them and use the class files they contain.
- You don't have to mention these subdirectories in a classpath statement since searching this directory is a function that's built right into java.
- If you see something like **JAVA_HOME** or **$JAVA** in an exam question it just means...

Example:
Simulate a scenario in which there is a:
- JAR file exttest1.jar which contains a
- class Cube which contains a static method cu which finds cube of an integer.
- Place the jar file in **...jre/lib/ext** and use it appropriately

Solution:

❖ **Step 1:** Create a file Cube.java as shown below in the current folder.

```
public class Cube
{
public static int cu(int x)
{       return x*x*x ;       }
}
```

❖ **Step 2:** Using command prompt go to current folder and give the command
javac Cube.java
This command creates Cube.class in the current folder.

❖ **Step 3:** Create JAR file by giving command: **jar -cfv exttest1.jar Cube.class**
added manifest
adding: Cube.class(in = 238) (out = 185) (deflated 22%)

❖ **Step 4: Place exttest1.jar in folder ...jre/lib/ext**

❖ **Step 5:** In the current folder create a file FindCubeExt.java

```
class FindCubeExt
{
public static void main(String args[])
{
 int c = Integer.parseInt(args[0]);
 c = Cube.cu(c) ;
 System.out.println(c) ;
} }
```

Compile Command : **javac FindCubeExt.java**
Run Command : **java FindCubeExt 4**

Note: classpath (-cp) is not mentioned while compiling and executing FindCubeExt, since the jar file containing Cube class is kept in ...jre/lib/ext

Object Class

1. In java class **O**bject (O capital) is a predefined class. It is the superclass of all other classes.
2. In other words, every class we create or use in java is the subclass of Object.
3. Hence, methods of class Object are inherited by all other classes.
4. Moreover, a reference variable of type Object can refer to an object of any other class. Also, since arrays are implemented as classes, a variable of type Object can also refer to any array.
5. Object defines the following methods:

Method	Purpose
String toString()	Returns a string that describes the object.
boolean equals(Object object)	Determines whether one object is equal to another.
int hashCode()	Returns the hash code associated with the invoking object.
void finalize()	Called before an unused object is recycled.
Object clone()	Creates a new object that is the same as the object being cloned.
Class getClass()	Obtains the class of an object at run time.
void notify()	Resumes execution of a thread waiting on the invoking object.
void notifyAll()	Resumes execution of all threads waiting on the invoking object.
void wait() void wait(long milliseconds) void wait(long milliseconds, int nanosecond)	Waits on another thread of execution.

getClass(), notify(), notifyAll() and wait() are final and hence cannot be overridden.

equals() and hashCode() are public and non-final and hence can be overridden.

toString()

1. The toString() method returns a string that contains a description of the object on which it is called.
2. Also this method is automatically called when an object is output using println().
3. Many classes override this method. Doing so allows them to tailor a description specifically for the types of objects that they create.

Example 1: Using toString() of Object

```
class Empty
{ }

class ToString1
{
public static void main(String args[])
{
   String  s1;

   Empty  e1=new Empty();
   s1 = e1.toString();
   System.out.println(s1);

   System.out.println(e1);

   Object  ref;
   ref = e1;
   s1=ref.toString();
   System.out.println(s1);
}
}
```

class name — address — hexadecimal address of referring object

O/P:-
Empty@6d06d69c
Empty@6d06d69c
Empty@6d06d69c

same words

Polymorphic Ref [object is superclass of class Empty]

Example 2: Overriding toString()

```
class Empty
{
     public String toString()
     { return  "I am Empty" ; }
}

class ToString2
{
public static void main(String args[])
{
   String  s1;

   Empty  e1=new Empty();
   System.out.println(e1);
}
}
```

overrides the super class "object"

O/P: I am Empty

equals()

[if both obj belongs to same class]

1. This method checks whether the object **references** of the two objects are equal or not.
2. If the two references are equal then the method returns true else it returns false.
3. However, if the objects being compared belong to class **String** or a **Wrapper** class then equals() method compares the **content** of the two objects.
4. If you want equals() method to compare content of two objects (for objects other than String and Wrapper) then you need to override it.

Example 1: Using equals() of Object

o/p:- eobj1 not equal to eobj2
iobj1 equals iobj2
s1 equals s2

```
class Equals1
{
        private int e ;

        public Equals1(int x)
        { e = x ; }
}

class EqualsTest1
{
        public static void main(String args[ ])
        {
        Equals1 eobj1 = new Equals1(5);
        Equals1 eobj2 = new Equals1(5);

        if(eobj1.equals(eobj2))
                System.out.println("eobj1 equals eobj2");
        else
                System.out.println("eobj1 not equal to eobj2");

        Integer iobj1=new Integer(6);
        Integer iobj2=new Integer(6);

        if(iobj1.equals(iobj2))
                System.out.println("iobj1 equals iobj2");
        else
                System.out.println("iobj1 not equal to iobj2");

        String s1 = new String("OCJP");
        String s2 = new String("OCJP");

        if(s1.equals(s2))
                System.out.println("s1 equals s2");
        else
                System.out.println("s1 not equal to s2");
        }
}
```

o/p:
~~eobj1 equals eobj2~~
~~eobj1 not equal to eobj2~~
iobj1 equals iobj2
~~s1 equals s2~~

eobj1 → e=5
eobj2 → e=5
iobj1 → 6
~~iobj1~~
iobj2 → 6
s1 → "OCJP"
s2 → "OCJP"

Wrapper class

Example 2: Overriding equals()

```
class Equals2
{
    private int e ;

    public Equals2(int x)
    { e = x ; }

    public int getValue()
    { return e ;}

    public boolean equals(Object o)
    {
    if( (o instanceof Equals2) && ((Equals2)o).getValue() == e )
            return true ;
    else
            return false ;
    }
}

class EqualsTest2
{
    public static void main(String args[ ])
    {
    Equals2 eobj1 = new Equals2(5);
    Equals2 eobj2 = new Equals2(5);

    if(eobj1.equals(eobj2))
            System.out.println("eobj1 equals eobj2");
    else
            System.out.println("eobj1 not equal to eobj2");
    }
}
```

class Object { public equals(Object o) { } }

Hidden class

Used defined equals(), this equals checks contents of obj

Method overriding of equals() of class object

o is eobj2

Class Object

e of eobj2

This is e (instance var) of eobj1

same with little diff

o/p:- eobj1 equals eobj2

eobj1 → e=5

eobj2 → e=5

o (Object)

Note: ① The method equals receives the ~~Object~~ parameter eobj2 and stores it in O of class Object. To invoke the method 'getValue()', we need to typecast 'O' again to Equals2, since otherwise a variable of Superclass cannot invoke ~~obj~~ a method ~~of s~~ which is exclusively present in subclass.

② We could have kept parameter O of datatype 'Equals2', but then it wouldn't be method overriding

Web:- (Purpose of 1st condition) In this program the first condition will be false and the short circuit && will avoid execution of 2nd condition thereby avoiding Class Cast Exception

Example 3: Importance of first condition

```
class Useless
{ public int u = 5 ; }

class Equals3
{
        private int e ;

        public Equals3(int x)     { e = x ; }

        public int getValue()     { return e ; }

        public boolean equals(Object o)
        {
        if( (o instanceof Equals3) && ((Equals3)o).getValue() == e)
                return true ;
        else
                return false ;
        }
}

class EqualsTest3
{
        public static void main(String args[ ])
        {
        Equals3 eobj1 = new Equals3(5);
        Useless uobj2 = new Useless();

        if(eobj1.equals(uobj2))
                System.out.println("eobj1 equals uobj2");
        else
                System.out.println("eobj1 not equal to uobj2");
        }
}
```

The equals() Contract

Pulled straight from the Java docs, the equals() contract says

1. It is **reflexive**. For any reference value x, x.equals(x) should return true.

2. It is **symmetric**. For any reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.

3. It is **transitive**. For any reference values x, y and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) must return true.

4. It is **consistent**. For any reference values x and y, multiple invocations of x.equals(y) should consistently return true or consistently return false, provided no information used in equals() comparisons on the object is modified.

5. For any non-null reference value x, x.equals(null) should return false.

hashCode()

1. Every object in java is assigned a unique integer ID called its hashcode.
2. This hashcode is used by java to store the object in a data structure. The same hashcode is then used to retrieve the object from data structure.
3. The hashCode() method returns the hashcode of the invoking object.

Example 1: Using hashCode() of Object

```
class HashCodeTest1
{
    public static void main(String args[ ])
    {
        HashCodeTest1 hobj1 = new HashCodeTest1();
        HashCodeTest1 hobj2 = new HashCodeTest1();
        HashCodeTest1 hobj3 = hobj2 ;

        System.out.println("Hashcode of hobj1 = "+hobj1.hashCode());
        System.out.println("Hashcode of hobj2 = "+hobj2.hashCode());
        System.out.println("Hashcode of hobj3 = "+hobj3.hashCode());
    }
}
```

O/P :-
Hashcode of hobj 1 = 1932156700
Hashcode of hobj 2 = 645321789
Hash code of hobj 3 = 645321789

The hashCode() Contract

Pulled straight from the Java docs, the hashCode() contract says

- Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode() method must consistently return the same integer, provided that no information used in equals() comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.

- If two objects are equal according to the equals(Object) method, then calling the hashCode() method on each of the two objects must produce the same integer result.

- It is NOT required that if two objects are unequal according to the equals(Object) method, then calling the hashCode() method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables.

	Condition	Required	Not Required (But Allowed)
1	x.equals(y) == true	x.hashCode() == y.hashCode()	
2	x.equals(y) == false	No hashCode() requirements	
3	x.hashCode() == y.hashCode()		x.equals(y) == true
4	x.hashCode() != y.hashCode()	x.equals(y) == false	-

Example 2: Overriding hashCode()

```
class HashCode2
{
        private int e ;

        public HashCode2(int x)
        { e = x ; }

        public int getValue()
        { return e ;}

        public boolean equals(Object o)
        {
        if( (o instanceof HashCode2) && ((HashCode2)o).getValue() == e )
                return true ;
        else
                return false ;
        }

        public int hashCode()
        { return e * 17 ; }
}

class HashCodeTest2
{
        public static void main(String args[ ])
        {
        HashCode2 hobj1 = new HashCode2(5);
        HashCode2 hobj2 = new HashCode2(5);
        HashCode2 hobj3 = new HashCode2(6);

        System.out.println("Hashcode of hobj1 = "+hobj1.hashCode());
        System.out.println("Hashcode of hobj2 = "+hobj2.hashCode());
        System.out.println("Hashcode of hobj3 = "+hobj3.hashCode());

        if(hobj1.equals(hobj2))
                System.out.println("hobj1 equals hobj2");
        else
                System.out.println("hobj1 not equal to hobj2");

        if(hobj1.equals(hobj3))
                System.out.println("hobj1 equals hobj3");
        else
                System.out.println("hobj1 not equal to hobj3");
        }
}
```

O/P : 85
85
102
hobj1 equals hobj2
hobj1 not equal to hobj3

hobj1 → e=5
hobj2 → e=5
hobj3 → e=6

→ It satisfies hashcode contract

Example 3: VALID & APPROPRIATE but horribly INEFFICIENT hashCode() method

```
class HashCode3
{
        private int e ;

        public HashCode3(int x)
        { e = x ; }

        public int getValue()
        { return e ;}

        public boolean equals(Object o)
        {
        if( (o instanceof HashCode3) &&  ((HashCode3)o).getValue() == e )
                return true ;
        else
                return false ;
        }

        public int hashCode()
        { return 1760 ; }
}

class HashCodeTest3
{
        public static void main(String args[ ])
        {
        HashCode3 hobj1 = new HashCode3(5);
        HashCode3 hobj2 = new HashCode3(6);

        System.out.println("Hashcode of hobj1 = "+hobj1.hashCode());
        System.out.println("Hashcode of hobj2 = "+hobj2.hashCode());

        if(hobj1.equals(hobj2))
                System.out.println("hobj1 equals hobj2");
        else
                System.out.println("hobj1 not equal to hobj2");
        }
}
```

← (row 2 of table)

hobj1 → e=5

hobj2 → e=6

O/P :- 1760
1760
hobj1 not equal to hobj2

→ It obeys hashcode contract, but we shall avoid this in real world bcz it is inefficient.

Garbage Collection

Need of Garbage Collection

1. Consider a program which reads data from the user, stores it into some sort of collection (like array, vector etc) in memory, performs some operations on the data and then writes the data into the database.
2. After the data is written into the database, the collection that stored the data temporarily must be emptied of old data or deleted and re-created before processing the next batch.
3. This operation might be performed thousands of times, and in languages like C or C++ that do not offer automatic garbage collection(memory de-allocation), a small flaw in the logic that manually empties or deletes the collection can allow small amounts of memory to be improperly reclaimed or lost forever.
4. These small losses are called memory leaks, and over many thousands of iterations they can make enough memory inaccessible. Finally the program will eventually crash for lack of memory.
5. Creating code that performs manual memory management cleanly and thoroughly is a complex task.
6. Java handles memory de-allocation completely automatically for the programmer. The mechanism that accomplishes this automatic memory de-allocation is called Garbage Collection.

How Garbage Collector Works ?

1. The heap is that part of memory where Java objects live, and it's the one and only part of memory that is in any way involved in the garbage collection process. So, all of garbage collection revolves around making sure that the heap has as much free space as possible.
2. When the garbage collector runs, its purpose is to find and delete those objects that cannot be "reached" and are thus eligible for garbage collection. An object is eligible for garbage collection when no live thread can access it.
3. The garbage collector is under the control of the JVM. JVM decides when to run the garbage collector. Garbage collection system of java occurs at irregular intervals during the execution of the program.
4. It cannot be specifically told when the garbage collection will take place. Moreover, it is also possible that Garbage Collection does not take place at all and memory allocated to the object is never de-allocated.

finalize() method

1. Java provides a mechanism that lets you run some code just before your object is deleted by the garbage collector. This code is located in a method named finalize() that all classes inherit from class object. The garbage collector automatically executes this method before deleting the object.
2. On the surface, this sounds like a great idea. The problem is that you can never count on the garbage collector to delete an object. So, your class's overridden finalize() method may never run.
3. Hence it is not advisable to put any essential code into your finalize() method. In fact, it is recommended that in general you don't override finalize() at all.

Tricky concepts about finalize()

There are a couple of concepts concerning finalize() that you need to remember:

1. For any given object, finalize() will be called ***only once*** (at most) by the garbage collector.
2. Calling finalize() can actually result in ***saving an object from deletion***. For example, in the finalize () method, you could write code that creates a new reference to the object which is about to be deleted, effectively ineligible-izing the object for garbage collection. The garbage collector can still process the object and delete it if it later becomes eligible for garbage collection. The garbage collector, however, will remember that for this object, ***finalize() already ran***, and it will not run finalize() again for the same object.

Writing code that Explicitly Makes Objects Eligible for Collection

a) Nulling a Reference

An object becomes eligible for garbage collection when there are no more reachable references to it. Obviously, if there are no reachable references, it doesn't matter what happens to the object. The first way to remove a reference to an object is to set the reference variable that refers to the object to null.

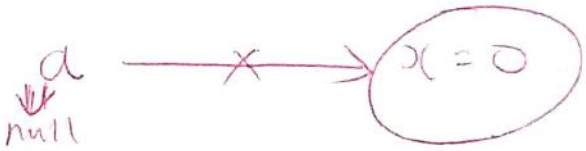
Example 1:

```
public class GarbageTruck   {
public static void main (String  []  args)
{
        StringBuffer sb = new StringBuffer("hello") ;
        System.out.println(sb);
        // The StringBuffer object is not eligible for garbage collection at this point
        sb = null;
        // Now the StringBuffer object is eligible for garbage collection
}    }
```

into the finalize() method

Example 2:

```
class GC
{
 private  int  x ;

 public void show()
 {    System.out.println(x);  }

 protected void finalize()
 {    System.out.println("Finalize Executes");  }
}
class GCMain
{
public static void main(String args[])
{
 GC a = new GC();
 a.show();
 a=null;
 System.out.println("About to exit main"); } }
```



O/P: 0
Finalize Executes
About to exit main

(or)

About to exit main
Finalize Executes

(or)

0
About to exit main

Garbage collector doesn't execute

Example 3: (using System.gc())

```
class GC2
{
 private int x ;

 public void show()
 {    System.out.println(x); }

 protected void finalize()
 {    System.out.println("Finalize Executes"); }
}

class GCMain2
{
public static void main(String args[])
{
 GC2 a = new GC2();
 a.show();
 a=null;
 System.gc();
 System.out.println("About to exit main");
 System.gc();
 System.gc();
}
}
```

(Predefined System class cmd gc() request to grab coll to deallocate mem)

(gc() executes only once for each object)

(In this code, System.gc() is 3 times written, but it only one of it will be performed. We can't say which one of it)

O/P:- 0
Finalize Executes
About to exit main

(or)

0
About to exit main
Finalize Executes

(This o/p come bcoz, GC could happen after S.o.P(About to exit main))

Example 4: (finalize is called explicitly)

```
class GC1
{
 private int x ;

 public void show()
 {
    System.out.println(x);
 }

 protected void finalize()
 {
    System.out.println("Finalize Executes");
 }
}

class GCMain1
{
public static void main(String args[])
{
 GC1 a = new GC1();
 a.show();
 a.finalize();
 System.out.println("About to exit main");
}
}
```

O/P:- 0
Finalize Executes
About to Exit main

(explicitly manually call finalize())

[Here a is not assigned null]

→ finalize() executes only once

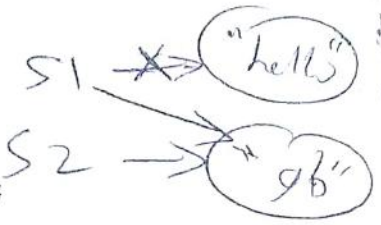
Sandeep J. Gupta (9821882868)

→ If we have called finalize() explicitly then GC will not execute finalize() again for that obj

b) Reassigning a Reference Variable

We can also decouple a reference variable from an object by setting the reference variable to refer to another object. Consider the following code:

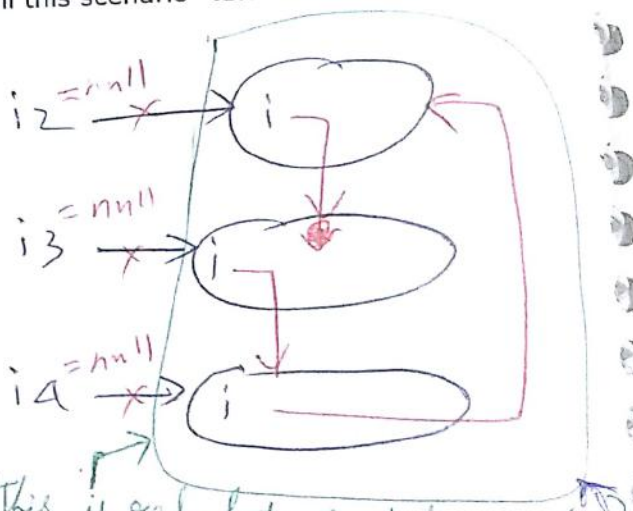
```
class GarbageTruck
{
   public static void main (String  []  args)
   {
        StringBuffer s1 = new StringBuffer("hello") ;
        StringBuffer s2 = new StringBuffer("goodbye") ;
        System.out.println(s1) ;
        // At this point "hello" is not eligible
        s1 = s2;  // Redirects s1 to refer to the "goodbye" object
        // Now "hello" is eligible for collection
   }
}
```



c) Islands of Isolation

1. There is another way in which objects can become eligible for garbage collection, even if they still have valid references! We call this scenario ***"islands of isolation."***
2. Examine the following code:

```
public class Island  {
     Island  i;
  public static void main( String [] args )  {

     Island i2 = new Island() ;
     Island i3 = new Island() ;
     Island i4 = new Island() ;

     i2.i = i3;     // i2 refers to i3
     i3.i = i4;     // i3 refers to i4
     i4.i = i2;     // i4 refers to i2

     i2 = null;
     i3 = null;
     i4 = null;
    // do complicated,memory intensive stuff
  }
}
```



This is redundant, as it doesn't have contact with outside world, so gc deallocates their mem

In this example Island class has an instance variable which is also a reference variable of type Island.

When the code reaches // do complicated, the three Island objects (previously known as i2,i3,and i4) have instance variables so that they refer to each other, but their links to the outside world (i2,i3 and i4) have been nulled.

These three objects constitute an "island of objects" and are eligible for garbage collection.

(They have address of each other but no outside world has their address)

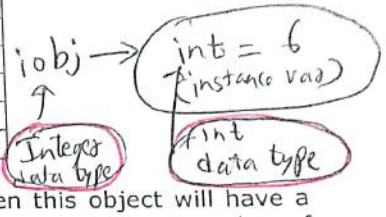
Wrapper Classes

1. A wrapper class is normally used for some kind of ***conversion***. Wrapper classes are contained in the java.lang package.
2. The following table shows primitive data types & the corresponding wrapper class.

(built in data type)

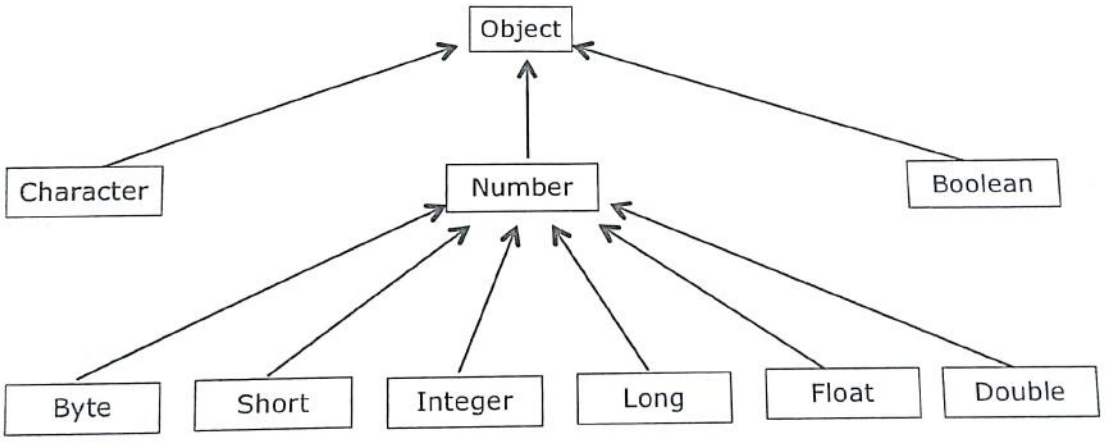
Primitive Type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Integer iobj = new Integer(6);



3. If we create an object of a wrapper class, say Integer, then this object will have a field (instance variable) of data type int. In this field we can store any value of primitive data type int. This is true for all wrapper classes.

4. Shown below is the wrapper class hierarchy.



5. Each wrapper class contains many methods. Each method will do some conversion. Shown below is a list of all such methods.

A) Constructors of wrapper classes

1. Each wrapper class (except Character) contains two constructors
   a) One that takes a value of primitive data type as its argument and
   b) another which takes a string as its argument.
2. Example:

```
class WrapperCons {
    public static void main(String args[ ])
    {
        Integer iobj1=new Integer(4);
        Integer iobj2=new Integer("6");
        System.out.println(iobj1+" "+iobj2);

        Double dobj1=new Double(4.2);
        Double dobj2=new Double("6.8");
        Double dobj3=new Double("6.8f");
        System.out.println(dobj1+" "+dobj2+" "+dobj3);

        Character cobj1=new Character('B');
        System.out.println(cobj1);

        Boolean bobj1=new Boolean(true);
        Boolean bobj2=new Boolean("FALSE"); // Uppercase allowed
        System.out.println(bobj1+" "+bobj2);

        Float fobj1=new Float(4.2);
        Float fobj2=new Float("6.8");
        Float fobj3=new Float("6.8f");
        System.out.println(fobj1+" "+fobj2+" "+fobj3);
        short s=4;
        Short sobj1=new Short(s); // error if we directly write 4
        Short sobj2=new Short("6");
        System.out.println(sobj1+" "+sobj2);

        byte by=4;
        Byte byobj1=new Byte(by); // error if we directly write 4
        Byte byobj2=new Byte("6");
        System.out.println(byobj1+" "+byobj2);

        Long lobj1=new Long(4);
        Long lobj2=new Long("6");
        System.out.println(lobj1+" "+lobj2);
    }
}
```

O/P:
```
4 6
4.2 6.8 6.8
B
true false
4.2 6.8 6.8
4 6
4 6
4 6
```

B) typeValue()

1. The method typeValue() actually refers to methods byteValue(), shortValue(), intValue(), longValue(), floatValue(), doubleValue(), charValue() and booleanValue().
2. This method converts a wrapper object to primitive type.
3. Example:

```
Double dobj1=new Double(6.8);
double d = dobj1.doubleValue();
Character cobj1=new Character('B');
char c = cobj1.charValue();
Boolean bobj1=new Boolean(true);
boolean b = bobj1.booleanValue();
```

C) valueOf()

1. valueOf() is a static method. It is present in all wrapper classes except Character.
2. This method takes string as an argument and converts it to a wrapper object.
3. Example:

```
String s1 = "24";
Integer iobj1 = Integer.valueOf(s1);
System.out.println(iobj1);
```

D) toString()

1. This method takes a wrapper object as an argument and converts it to a string.
2. Example:

```
Float fobj1 = new Float(25.8);
String s2 = fobj1.toString();
System.out.println(s2);
```

E) parseType()

1. The method parseType() actually refers to methods parseByte(), parseShort(), parseInt(), parseLong(), parseFloat() and parseDouble().
2. Character and Boolean classes do not have this method.
3. parseType() is a static method.
4. This method takes string as an argument and converts it to a numeric value of primitive data type.
5. The method can throw NumberFormatException if the string is not of expected format.
6. Example:

```
String s3 = "56";
long l = Long.parseLong(s3);
System.out.println(l);
```

F) toString() (converts Primitive to String)

1. This method takes a primitive type as an argument and converts it to a string.
2. Example:

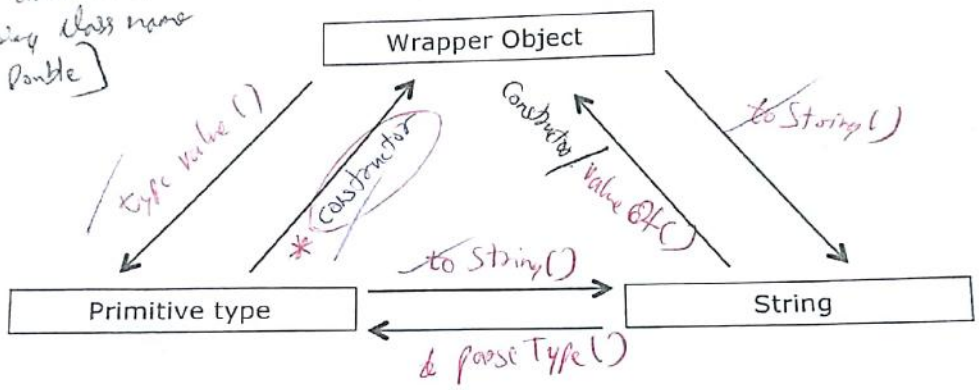
```
String s4 = Double.toString(2.4);
System.out.println(s4);
```

(Primitive data type double)

o/p = 2.4 ← String

[toString() is Static method as it is invoked using class name Double]

MORAL OF THE STORY



Autoboxing and Unboxing

1. When Java automatically converts a primitive type like int into corresponding wrapper class object e.g. Integer than its called **autoboxing** because primitive is boxed into wrapper class while the opposite is called **unboxing**, where an Integer object is converted automatically into primitive int.
2. This is the new feature of Java 5. Because of this feature, the programmer doesn't need to write the conversion code manually.
3. Example:

```
class BoxingExample1
{
 public static void main(String args[])
 {
  // Before Java 5
  Integer iobj1=new Integer(5);                    //Boxing
  int i1 = iobj1.intValue();
  System.out.println(iobj1+" "+i1);

  //Java 5 onwards
  Integer iobj2 = 6;                               // AutoBoxing
  int i2 = iobj2 ;                                 // Unboxing
  System.out.println(iobj2+" "+i2);
 }
}
```

4. Lets explore one more convenience which auto-boxing offers. In the old, pre-java 5 days, if we wanted to make a wrapper, unwrap it, use it and then rewrap it, we might do something like this:

```
Integer y = new Integer(567) ;      // wrapping
int x = y.intValue() ;              // unwrap it
x++ ;                               // use it
y = new Integer(x) ;                // rewrap it
System.out.println("y = " + y) ;    // print it
```

Now, with new and improved java 5, we may write

```
Integer y = new Integer(567) ;      // wrapping
y++;                                // Auto-unwrap, increment and rewrap
System.out.println("y = " + y) ;    // print it
```

Both examples produce the following output:
y = 568

The code appears to be using the post increment operator on an object reference variable. But it's simply a convenience. Behind the scenes, the compiler does the unboxing and reassignment for us.

How Autoboxing works with == and equals() ?

1. == checks whether two object references are equal or not.
2. equals() will check content of objects (for wrapper classes).
3. == will return true for two wrapper instances (created through autoboxing) if they have same primitive value and lies in any one of the following categories:
   a. Boolean
   b. Byte
   c. Character from \u0000 to \u007f (0 to 127)
   d. Short and Integer from -128 to +127.
4. However, if new is used for object creation then a new object will be always created.
5. When == is used to compare a primitive to a wrapper, the wrapper will be unwrapped and the comparison will be primitive to primitive.

```
class BoxingExample2
{
 public static void main(String args[])
 {
  Integer iobj1=new Integer(5);
  Integer iobj2=new Integer(5);

  System.out.println(iobj1==iobj2);          // Rule 1 , 4
  System.out.println(iobj1.equals(iobj2)); // Rule 2
  System.out.println();

  // Rule 3
  Short sobj1 = 127 ;
  Short sobj2 = 127 ;
  Short sobj3 = 128 ;
  Short sobj4 = 128 ;

  System.out.println(sobj1==sobj2);
  System.out.println(sobj1.equals(sobj2));
  System.out.println();

  System.out.println(sobj3==sobj4);
  System.out.println(sobj3.equals(sobj4));
  System.out.println();

  // Rule 5
  short s = 6 ;
  Short sobj5 = 6 ;
  System.out.println(s==sobj5);
  // System.out.println(s.equals(sobj5)); ERROR since s cannot invoke equals()
 }
}
```

Method Overloading with – Autoboxing , Widening & Var-args

- A given class may have multiple methods with same name and different argument lists. Of these methods, one method may support widening, other may support autoboxing and another may support var-args.
- The question is then " Whom does the compiler give importance?". The answer is

Widening beats boxing
Widening beats varargs
Boxing beats varargs

A) Example of Widening

```
1. class  MOLWid
2. {
3. static void check(byte  b) {System.out.print("byte ");}
4. static void check(int  i)    {System.out.print("int ");}
5. static void check(long  l)  {System.out.print("long ");}
6. static void check(float  f)  {System.out.print("float ");}
7. static void check(double  d)  {System.out.print("double ");}

   public static void main(String args[])
   {
       byte  b = 6 ;
       int  i = 7 ;
       long l = 8 ;
       float f = 2.4f ;
       double d = 4.2 ;

       check(b);
       check(i);
       check(l);
       check(f);
       check(d);
   }
   }
```

Output: byte int long float double

What is the output if statements 3 and 6 are removed ?

int int long double double

What is the output if statement 5 is removed ?

byte int float float double

What is the output if statement 7 is removed ?

Compilation fails (bcoz double can be stored only in double)

Widening takes place as shown below:

byte => short => int => long => float => double

(Here byte can be stored in all above data types)

B) Example where Widening Beats Boxing

```
1. class  MOLWidBox
2. {
3. static void check(Byte  bobj) {System.out.print("Boxing ");}
4. static void check(int  i)    {System.out.print("Widening ");}
5. // static void check(byte  b) {System.out.print("Same data type ");}

   public static void main(String args[])
   {
         byte  b = 6 ;
         check(b);
   }
   }
```

Output: Widening

What is the output if statement 5 is not a comment ? → Same data type

C) Example where Widening Beats Var-Args

```
class  MOLWidVar
{
 static void check(byte...  x) {System.out.print("Var-Args");}
 static void check(int  i , int  j)    {System.out.print("Widening ");}

 public static void main(String args[])
 {
      byte  b1 = 6 ;
      byte  b2 = 5 ;
      check(b1,b2);
 }
}
```

Output: Widening

D) Example where Boxing Beats Var-Args

```
1. class  MOLBoxVar
2. {
3. static void check(byte...  x) {System.out.print("Var-Args");}
4. static void check(Byte  bobj1 , Byte  bobj2)    {System.out.print("Boxing ");}
5. //static void check(Integer  iobj1 , Integer  iobj2) {System.out.print("Integer Boxing ");}

   public static void main(String args[])
   {
     byte  b1 = 6 ;
     byte  b2 = 5 ;
     check(b1,b2);
   }
   }
```

Output: Boxing

What is the output if statement 4 is removed and 5 is added ? ~~Integer Boxing~~ Var-Args

What is the output if statements 3 and 4 both are removed ? Error but 5 is there

There is no longest called Widened Boxing in Java (Integer)

Enumeration (enum)

(enum is also a class)

1. Enumeration in java is a special data type that contains fixed set of constants. It is normally used when we know in advance that a particular variable will always take a value from a small set of known values.
2. Lets say, there is a variable d whose value can be any day of the week. We may then create an enum like
   enum Day { SUNDAY, MONDAY, TUESDAY, WEDNESDAY,THURSDAY, FRIDAY, SATURDAY}
   and then we may declare a variable like: *Day d ;*
   Now d may take any value like *SUNDAY, MONDAY, TUESDAY, etc.*
3. In the above example, *SUNDAY, MONDAY, TUESDAY, etc.* are called enums and they are implicitly *final* and *static*. In other words, they are constants and that's why they are normally given in uppercase (although its not a rule).
4. The enum data type is actually a special kind of ***Java class*** which may contain constants, variables, constructors and methods. Java enums were added in Java 5.

Example 1:

[Here Pizza is a class (data type) user defined]

```
enum Pizza {SMALL, REGULAR, BIG} ;

class PizzaMain
{
 public static void main(String args[])
 {
      Pizza p ;
      p = Pizza.BIG ;
      // Pizza p = Pizza.BIG ;

      switch(p)
      {
        case SMALL: System.out.println("SMALL Pizza is for 1 person") ; break ;
        case REGULAR: System.out.println("REGULAR Pizza is for 2 persons") ; break ;
        case BIG: System.out.println("BIG Pizza is for 4 persons") ; break ;
      }

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

(enum constants)

o/p: Big Pizza is for 4 persons
2

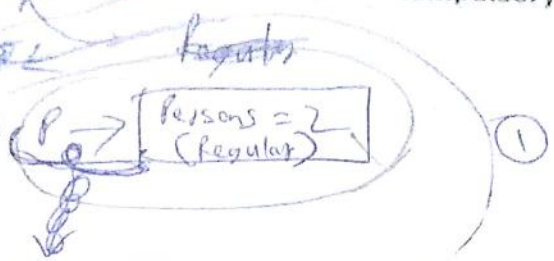
(gives the position no of 'p' of Pizza)

Note:

1. semicolon optional in first statement
2. enum cannot be protected or private. It can be public, but then file name should be same as enum name.
3. enum constants are final and static. Hence the follwing is invalid:
   Pizza.BIG = Pizza.SMALL;
   (because of final)
4. ordinal() gives associated position of each enum constant which starts from 0.

Example 2: enum with Constructor and member Variable

```
enum Pizza1
{
        SMALL(1), REGULAR(2), BIG(4) ; // semicolon compulsory now

        Pizza1(int pr)
        { persons = pr ; }

        public int persons ;
}

class Pizza1Main
{
 public static void main(String args[])
 {
        Pizza1 p = Pizza1.REGULAR ;

        System.out.println(p+" pizza is for "+p.persons+" persons ") ;
 }
}
```



Output: Regular pizza is for 2 persons

Note:

1. We can initialize enum constants with specific values. But for this to work we need to define a member variable and a constructor because **BIG(4)** is actually calling a constructor which accepts a value of data type int.

2. By default, any constructor defined within enum is private. Anything else reports error.

3. Normally the constructor is written as

```
Pizza1(int persons)
{ this.persons = persons ; }
```

4. Semicolon compulsory in *SMALL(1), REGULAR(2), BIG(4) ;*

5. Access specifier of *persons* can be anything, but if it is private, it cannot be accessed in other classes.

6. *SMALL(1), REGULAR(2), BIG(4) ;* should be the first line of enum. Otherwise compiler reports error.

Example 3: Multiple member Variables and Member Methods

(instance variables)

```
enum Pizza2
{
    SMALL(1,100.25), REGULAR(2,150.75), BIG(4,200.50) ;

    private int persons ;

    private double cost ;

    Pizza2(int persons , double cost)
    {
        this.persons = persons ;
        this.cost = cost ;
    }

    int getPersons()
    { return persons ; }

    double getCost()
    { return cost ; }
}

class Pizza2Main
{
 public static void main(String args[])
 {
  for(Pizza2 p : Pizza2.values())
     System.out.println(p+" pizza is for "+p.getPersons()+" persons and costs Rs."
                        +p.getCost()) ;
 }
}
```

P → [Persons = 1, cost = 100.25]

P → [Persons = 2, cost = 150.75]

P → [Persons = 4, cost = 200.5]

values() is predefined method which will return array of enums :- SMALL, REGULAR, BIG

Output:

SMALL Pizza is for 1 persons and costs Rs 100.25
REGULAR Pizza is for 2 persons and costs Rs 150.75
BIG Pizza is for 4 persons and costs Rs 200.50.

Note:

1) There are two member variables and that too private.
2) The constructor takes two parameters.
3) The enum has two methods.
4) values()

Java compiler automatically generates static method values() for every enum. values() method returns array of enum constants in the same order they are listed in enum declaration. Hence we can use values() to iterate over values of enum as shown in above program.

Example 4: enum in Class

```
class  Pizza3Main
{
 enum Pizza3 {SMALL, REGULAR, BIG} ;

 public static void main(String args[])
 {
      Pizza3 p1 = Pizza3.BIG ;
      Pizza3 p2 = Pizza3.SMALL ;

      System.out.println(p1+" "+p2) ;
 }
}
```

Output: BIG SMALL

Example 5: Similar to Example 3 but enum is in Class

```
class  Pizza4Main
{
 enum Pizza4
 {
      SMALL(1,100.25), REGULAR(2,150.75), BIG(4,200.50) ;

      private int persons ;

      private double cost ;

      Pizza4(int  persons , double  cost)
      {
            this.persons = persons ;
            this.cost = cost ;
      }

      int  getPersons()
      { return  persons ; }

      double  getCost()
      { return  cost ; }
}

public static void main(String args[])
{
     for(Pizza4 p : Pizza4.values())
        System.out.println(p+" pizza is for "+p.getPersons()+" persons and costs Rs."
                           +p.getCost()) ;

 }
}
```

Output: Same O/P as eg 3.

Example 6: enum defined in one class and used in another

```
class  Pizza5Class
{
  enum Pizza5 {SMALL, REGULAR, BIG} ;
}

class  Pizza5Main
{
  public static void main(String args[])
  {
      Pizza5Class.Pizza5 p1 = Pizza5Class.Pizza5.BIG   ; // Note change in syntax
      Pizza5Class.Pizza5 p2 = Pizza5Class.Pizza5.SMALL ;

      System.out.println(p1+" "+p2) ;
  }
}
```

Output: BIG SMALL

Example 7: enum with "Constant Specific Class Body"

```
enum PizzaC
{
      SMALL, REGULAR, BIG {

            String getMsg()
              { return  "U get a free drink !! " ; }
      } ;
      // Semicolon compulsory

    String getMsg()
    { return  "Sorry!! No free drink... :-) " ; }
}
class  CSCB
{
  public static void main(String args[])
  {
      PizzaC p1 = PizzaC.SMALL ;
      PizzaC p2 = PizzaC.REGULAR ;
      PizzaC p3 = PizzaC.BIG ;

      System.out.println(p1.getMsg()) ;
      System.out.println(p2.getMsg()) ;
      System.out.println(p3.getMsg()) ;
  }
}
```

Note:
*The **"constant specific class body"** is used when we want a particular constant to override a method defined in enum. In the above example, the "constant specific class body" defines a method getMsg() which overrides the normal getMsg() method for the constant BIG.*

Example 8: How ==, != and equals() work with enum constants ?

```
class  EnumEquals
{
 enum Color {RED, BLUE} ;

 public static void main(String args[])
 {
      Color c1 = Color.RED ;
      Color c2 = Color.RED ;
      Color c3 = Color.BLUE ;

      System.out.println(c1==c2) ;
      System.out.println(c1.equals(c2)) ;
      System.out.println(c1!=c2) ;

      System.out.println(c1==c3) ;
      System.out.println(c1.equals(c3)) ;

      c1 = c3 ;
      System.out.println(c1==c2) ;

      // System.out.println(c1<=c3) ; ERROR.
 }
}
```

Note:

1) == and equals() work the same way.
2) c1 = c3 is valid
3) Only == , != and equals() allowed. Operators like <, <=, >, >= not allowed.

Example 9: Invalid Syntax

```
class  Invalid
{
 public static void main(String args[])
 {
      enum  Seasons {SUMMER, MONSOON, WINTER} ;
 }
}
```

Note:
We cant declare enum inside method. Compiler reports following error:
"enum types must not be local"

Nested Classes

Just as classes have member variables and methods, a class can also have member classes called Nested Classes. Nested classes can be of the following four types:

A. Inner Classes
B. Method-Local Inner Classes
C. Anonymous Inner Classes
D. Static Nested Classes

A) Inner Classes

An Inner Class is sometimes also called 'Regular Inner Class'. We define an inner class within the curly braces of the outer class. For example,

```
class MyOuter
{
    class MyInner { }
}
```

And if you compile it as : javac MyOuter.java

you'll end up with two class files:

```
MyOuter.class
MyOuter$MyInner.class
```

The inner class is still, in the end, a separate class, so a separate class file is generated for it.
We cannot execute an inner class by writing the command: java MyOuter$MyInner
The only way you can access the inner class is through a live instance of the outer class.

To create an instance of an inner class, you must have an instance of the outer class to tie to the inner class.

There are no exceptions to this rule:

An inner class instance can never stand alone without a direct relationship to an instance of the outer class.

Instantiating (Creating an object of) Inner Class from within the Outer (enclosing) Class

Most often, it is the outer class that creates instances of the inner class, since it is usually the outer class wanting to use the inner instance as a helper for its own personal use.

Example:

```
class A1
{
 private int x = 6 ;

 class B1
 {
       public void show()
       {
       } System.out.println(x) ;
 }

 public void show()
 {
          B1 b = new B1() ;
           b.show();
 }

 public static void main(String args[])
 { new A1().show() ; }
}
```

Object of inner class is created inside non-static method of outer (enclosing) A1 class.

Output: 6

Compilation creates:
A1.class
A1$B1.class

Instantiating (creating an object of) inner class from within static method of outer class:

```
class  A2
{
 private int x = 6 ;

 class B2
 {
      public void show()
      {
            System.out.println(x) ;
      }
}

public static void main(String args[])
{
    // B2 b = new B2() ; ERROR - Cant create object this way in static method

     A2  a = new  A2();
     A2.B2  b = a.new B2() ;
     b.show();

     A2.B2  c =  new A2().new B2() ;
     c.show();
}
}
```

Only a Syntax, no logic to create a obj of inner class from static method of outer class

(object on fly)

Output:

```
6
6
```

We have to create obj of outer class also, to create a object of inner class which is inside static method

Instantiating (creating an object of) an inner class from outside of outer class

```
class A3
{
 private int x = 6 ;

 class B3
 {
      public void show()
      {
           System.out.println(x) ;
      }
 }
}

class C3
{
 public static void main(String args[])
 {
    // B3 b = new B3(); ERROR. Cant create object this way from outside of outer class

     A3  a = new  A3();
     A3.B3  b =  a.new B3() ;
     b.show();

     A3.B3  c =  new A3().new B3() ;
     c.show();

    // Note that the main looks similar to above example
 }
}
```

Output:

```
6
6
```

Compilation creates:

C3.class
A3.class
A3$B3.class

MORAL OF THE STORY:

1. The inner class name should be used in the normal way from ***inside instance code (non-static code)*** of outer class. For eg.

   ```
   InnerClass  i = new InnerClass();
   ```

2. From ***outside the outer class*** and from ***inside static code of outer class***, the inner class name must include outer class name. For eg.

   ```
   Outer  a = new  Outer();
   Outer. Inner  b =  a.new  Inner () ;
   ```

   OR

   ```
   Outer.Inner  c =  new Outer().new Inner () ;
   ```

→ **Usage of keyword this**

```
class  A4
{
 class B4
 {
      public void show()
      {
            System.out.println(this) ;
            System.out.println(A4.this) ;
      }
 }
 public static void main(String args[])
 {
     A4.B4  b =  new A4().new B4() ;
     b.show();
 }
}
```

Output:

```
A4$B4@6d06d69c
A4@7852e922
```

1. The keyword *this* can be used only from within instance code. In other words, not within static code. The *this* keyword is a reference to the currently executing (invoking) object i.e. the object whose reference was used to invoke the currently running method.

2. Within an inner class code, the *this* reference refers to the instance of the inner class, as you'd probably expect, since *this* always refers to the currently executing object.

3. But what if the inner class code wants an explicit reference to the outer class instance that the inner instance is tied to.

4. To reference the "outer this" (the outer class instance) from within the inner class code, use NameofOuterClass.this (example, MyOuter.this).

→ **Modifiers Applicable to Inner Classes**

A regular inner class is a member of the outer class just as instance variables and methods are, so the following modifiers can be applied to an inner class:

- final
- abstract
- public
- private
- protected
- static- but static turns it into a static nested class, not an inner class
- strictfp

B) Method - Local Inner Classes

1. A regular inner class is scoped inside another class's curly braces, but outside any method code (in other words, at the same level that an instance variable is declared).
2. But you can also define an inner class within a method. This class will be now called "Method-Local Inner Class". To use this class, you must make an instance of it somewhere within the method but ***below*** the inner class definition (or the compiler won't be able to find the inner class).
3. A method-local inner class can be instantiated only within the method where the inner class is defined. In other words, no other code running in any other method – inside or outside the outer class - can ever instantiate the method–local inner class.
4. Like regular inner class objects, the method–local inner class object shares a special relationship with the enclosing (outer) class object and can access its private (or any other) members.
5. However, the inner class object cannot use the local variables of the method the inner class is in, unless the local variables are marked ***final***.

Example 1:

```
class C1
{
 private int x = 6 ;

 public void show()
 {
  final int y = 5 ; // ERROR if final absent

  class D1
  {
   public void show()
   {
        System.out.println(x) ;
        System.out.println(y) ;
   }
  }

  D1 d = new D1(); // Must come after class D1
  d.show();
 }

 public static void main(String args[])
 {
  new C1().show() ;
  // D1 d = new D1(); ERROR. Cant use local class here. Its scope is enclosing method.
 }
}
```

Output: 6
5

Compilation creates:

C1.class
C1$1D1.class

Note: If y is named x, then o/p is 5 5. Hence local variable is given importance.

Example 2:

```
class C2
{
 private static int x = 6 ;

 public static void show()
 {
  final int y = 5 ;

  class D1
  {
      public void show()
      {
            System.out.println(x) ;
            System.out.println(y) ;
      }
  }

  D1 d = new D1();
  d.show();
 }

 public static void main(String args[])
 {
     new C2().show() ;
 }
}
```

Output:

```
6
5
```

Note:
If outer show() is static, then even x should be static. Otherwise error.
We cannot use static anywhere in Method-Local Inner Class.
Remember static is never used **WITHIN** method.

Modifiers Applicable to Method-Local Inner Classes

The same rules apply to method-local inner classes as to local variable declarations.

You **cannot** mark a method–local inner class as:
public, private, protected, static, transient, and the like.

The only modifiers you **can** apply to a method–local inner class are:
abstract and final (but never both at the same time).

Exam Tip:
A local class declared in a static method has access to only static members of the enclosing class.

V-Imp

C) Anonymous Inner Classes (nameless class)

i) Anonymous Inner Classes – Flavor One

So far, we've looked at defining a class within an enclosing class (a regular inner class) and within a method (a method-local inner class).

Finally, we're going to look at the most unusual syntax you might ever see in Java: inner classes declared without any class name at all (hence, the word anonymous).

Example 1:

Check out the following legal-but-strange code:

```
class Popcorn
{
   public void pop ()
   {
     System.out.println("popcorn");
   }
}
class Food
{
   public static void main(String args[])
   {
      Popcorn p1 = new Popcorn() ;
      p1.pop() ;

      Popcorn p = new Popcorn()  {
         public void pop()  {
            System.out.println("anonymous popcorn") ;
         }
      };

      p.pop() ;
   }
}
```

This is anynoma class which is subclass of class Popcorn

Anonymous class

O/P:- popcorn
anonymous popcorn

obj of subclass of class Popcorn

Let's look at the preceding code:

We define two classes:Popcorn and Food. Popcorn has one method: pop().
Food has a main method which has variables p1 and p, declared as type Popcorn.

And here's the big thing to get:

The Popcorn reference variable p refers not to an instance of Popcorn, but to ***an instance of an anonymous (unnamed) subclass*** of Popcorn.

Let's look at just the anonymous class code:

```
1. // Some Code here

2. Popcorn p = new Popcorn() {
3.    public void pop() {
4.       System.out.println("anonymous popcorn") ;
5.    }
6. };

7. // Some Code here
```

Line 2
Line 2 starts out as a variable declaration of type Popcorn. But **instead** of looking like this: Popcorn p = new Popcorn(); // notice the semicolon at the end
there's a curly opening brace at the end of line 2.
Popcorn p = new popcorn() { // a curly brace, not a semicolon
You can read line 2 as saying,
Declare a reference variable, p, of type Popcorn. Then declare a new class that has no name but that is a ***subclass of Popcorn***. And then put the curly brace that opens the class definition.

Line 3
Line 3, then, is actually the first statement within the new class definition. And what is it doing? Overriding the pop() method of the superclass Popcorn. This is the whole point of making an anonymous inner class - to override one or more methods of the superclass (Or to implement methods of an interface).

Line 4
Line 4 is the first (and in this case only) statement within the overriding pop() method. Nothing special there.

Line 5
Line 5 is the closing curly brace of the pop() method. Again nothing special.

Line 6
Here's where you have to pay attention. Line 6 includes a curly brace closing off the anonymous class definition (it's the companion brace to the one on line 2), but there's more! Line 6 also has the semicolon that ends the statement started on line 2 - the statement where it all began - the statement declaring and initializing the Popcorn reference variable. And what you're left with is a Popcorn reference to a brand new instance of a brand new, just-in-time, anonymous subclass of Popcorn.

Exam Tip:

The closing semicolon is hard to spot.

Watch for code like this which is **incorrect**:

```
2. Popcorn p = new Popcorn()  {
3.    public void pop()  {
4.       System.out.println("anonymous popcorn");
5.    }
6. }            // Missing semicolon needed to end statement started on Line 2
7. Foo f = new Foo();
```

ii) Anonymous Inner Classes – Flavor Two

The only difference between flavor one and flavor two is that flavour one creates an anonymous subclass of the specified class type, whereas flavour two creates an anonymous implementer of the specified interface type.

Example:

```
interface Cookable
{ public void cook(); }

class Foody
{       public static void main(String args[])
        {
           final String  s = "Anonymous" ;
           Cookable c = new Cookable() {
              public void cook() {
                  System.out.println(s+" cookable implementer");
              }
           };
           c.cook();
        }
}
```

Output: anonymous Cookable implementer.

Cookable is super interface of Anonymous class i.e anonymous class is subclass of interface Cookable

method overriding

we cannot create obj of interface, but we can create a obj of subclass of interface

The preceding code, like the Popcorn example, still creates an instance of an anonymous inner class, but this time, the new just-in-time class is an implementer of the Cookable interface. And note that is the only time you will ever see the syntax: ***new Cookable()*** where Cookable is an interface not a class.

Think about it: You can't instantiate an interface, yet that's what the code looks like it's doing. But, of course, it's not instantiating a Cookable object - it's creating an instance of a new anonymous implementer of Cookable.

You can read the line: Cookable c = new Cookable() {
as,
"Declare a reference variable of type Cookable that, obviously, will refer to an object of a class that implements the cookable interface".

One more thing to keep in mind about anonymous interface implementers is that they can implement only one interface.
In fact, an anonymous inner class can't even extend a class and implement an interface at the same time. The inner class has to choose either to be a subclass of a named class or to implement a single interface.

Exam Tip:
Don't be fooled by any attempts to instantiate an interface except in the case of an anonymous inner class. The following is not legal:
Runnable r = new Runnable(); // can't instantiate interface
whereas the following is legal, because it's instantiating an implementer of the Runnable interface
Runnable r = new Runnable() { // curly brace, not semicolon
  public void run() { }
};

Note: Anonymous class can access local final variable.

iii) Flavour Three - Argument-Defined Anonymous Inner Classes

```
interface Foo
{ void show() ; }

class Y1
{
      public void display(Foo F)
      { System.out.println("Class Y1") ; }
}

class X1
{
 public static void main(String args[])
 {
      Y1 y = new Y1() ;

      // Line 1

      y.display( new Foo() {
                        public void show() { }
                  } ) ;
 }
}
```

Output: class Y1

Note that till line 1 no variable of type Foo exists. Now below line 1 we call method display() and pass an instance of Foo using new. But wait!! We haven't sub-classed Foo till now.

Hence right then and there we create an anonymous class with method show() in it. It is this anonymous class whose instance is created and then passed to display().

D) Static Nested Classes

A static nested class is simply a class that's static member of the enclosing class. The static modifier in this case says that the nested class is a static member of the outer class. That means it can be accessed without having an instance of the outer class.

Example: Instantiating and Using static nested class from within enclosing class and non-enclosing class

```
class S1 (outer class)
{
 public static int x = 5 ;
 static class S2 (inner static class)
 {
     public void show()
     {     System.out.println(x) ;  }
 }
}

class Static (outer class)
{
 public static int x = 6 ;
 static class S3 (inner static class)
 {
     public void show()
     {     System.out.println(x) ;  }
 }

 public static void main(String args[])
 {
   S1.S2 a1 = new S1.S2() ;  // S2 a1 = new S2() ; is invalid
    a1.show();

    S3 a2 = new S3() ; // Static.S3 a2 = new Static.S3() ; is also valid
    a2.show() ;
 }
}
```

In inner class concept of HAS A we write obj here but here we write class name

No outer class name to be written, but we can write it here, we are creating in same class S3 Var a2

Output:

5
6

Compilation creates:

A1.class
A1$B1.class

Exam Tip:

Just as a static method does not have access to the instance variables and nonstatic methods of the class, a static nested class does not have access to the instance variables and nonstatic methods of the outer class. Look for static nested classes with code that behaves like a nonstatic (regular inner) class.

Modifiers Applicable to Static Nested Classes

final abstract public private protected strictfp static