**/\***

**\*\***

**\*\* JAVA 8 – Quick reference – 30-Sep-2018**

**\*\***

**\*/**

**Points to remember in the exam –**

* Always check the syntax first (especially when there is a “None of the above” or “Does not compile”)
* Be careful with final variables.
* Check the scope of variables (especially in the ones declared in try block)
* Pay attention to throw and throws
* .length for array; .length() for string

 **Float and double - IEEE 754**

|  |  |
| --- | --- |
| **Data Type** | **Default Value (for fields)** |
| byte | 0 |
| short | 0 |
| int | 0 |
| long | 0L |
| float | 0.0f |
| double | 0.0d |
| char | '\u0000' |
| String (or any object) | null |
| boolean | false |



**^ operator can only be used with Boolean**

**Identifiers –**

$123 Correct

\_123 Correct

Ab$c Correct

2BC Wrong

2abx Wrong

int Wrong

new Wrong

\_abc\_ Correct

abcX Correct

**Array-**

int[]blue = new int[2];

int[2]blue = new int[2]; 🡨 Wrong

int[] blue = new int[]; 🡨 wrong

int[] blue = new int[2];

int ary[] = new int[] {1,2,3,4};

int a[] = {1,2,3};

int ary[] = new int[4] {1,2,3,4}; 🡨 Wrong

int ary[] = new int[0] {}; 🡨Wrong

int ary[] = new int[] {};

int[] blue, red = new int[2]; 🡨 blue and red both are arrays

int blue[], red = new int[2]; 🡨 Wrong

int blue[], red = new int[2]; 🡨 blue is array, red is int variable; since there is no initialization – no issues

int blue[], red[] = new int[2];

int[][] blue = new int[2][2];

int[] blue[] = new int[2][2];

int[][] blue = new int[2,2]; 🡨 wrong

int[][] a, b = new int[2][1]; 🡨a and b both are 2D arrays

int[][] ary = new int[3][];

int ary[][] = new int[3][4];

int ary[][] = new int[][] {{1,2,3},{4,5,6}};

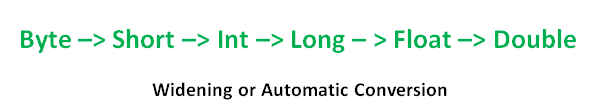
**switch Case –**

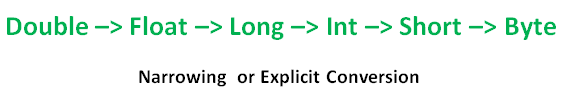
Works with ---

int, convertible to int(byte, short, char), strings and enums

Character, Byte, Short, and Integer

**Type Conversion –**





**Example- Explicit cast required**



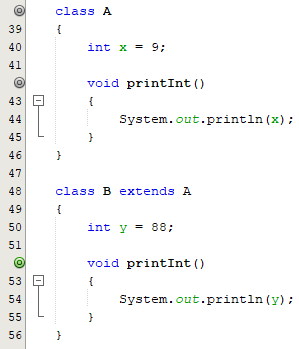
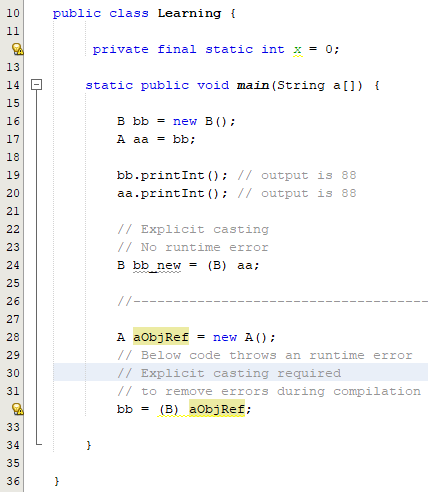
casting a reference variable doesn’t touch the object it refers to, but only labels this object in another way, expanding or narrowing opportunities to work with it. Upcasting narrows the list of methods and properties available to this object, and downcasting can extend it.

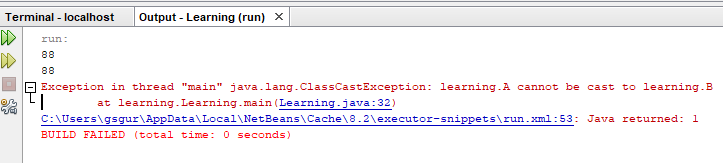
Upcasting – Narrowing – Casting to a super class – No cast required

Downcasting – Widening – Casting to a sub class – Explicit

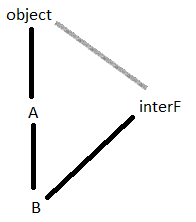
* Downcasting is necessary to gain access to members specific to subclass
* Downcasting is done using cast operator
* To downcast an object safely, we need instanceof operator
* If the real object doesn’t match the type we downcast to, then ClassCastException will be thrown at runtime

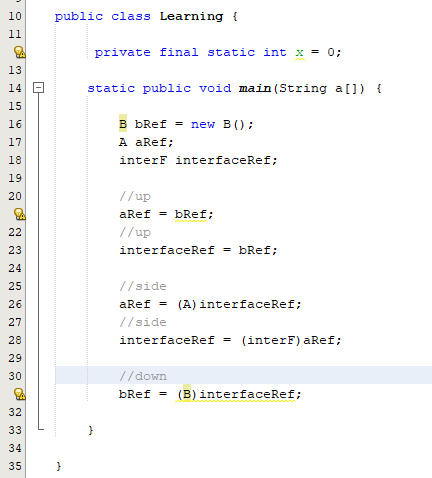
"If a subclass object is assigned to a super class object, the super class object will call subclass overridden method". This is an important rule which leads to dynamic polymorphism. This is not the case when it comes to hiding.





**Example –**

Downwards and sideways casting needs to be done explicitly**. **



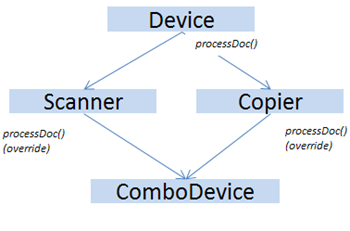
**class –**

* Only one public class in one .java file (file name same as the class name)

Inheritance –

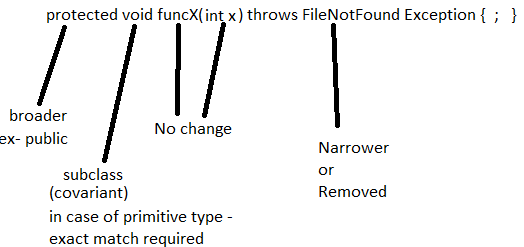
* Java does not support multiple inheritance in a single subclass. (To avoid diamond problem)
* Diamond problem can still occure with interfaces in java
* Order of constructor call – Super class to subclass

**Diamond problem –**

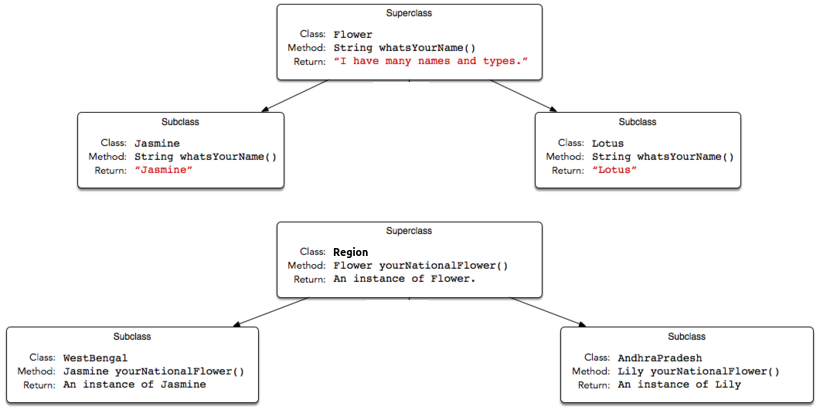


**Method overriding - Dynamic method dispatch (Run-time polymorphism)-**

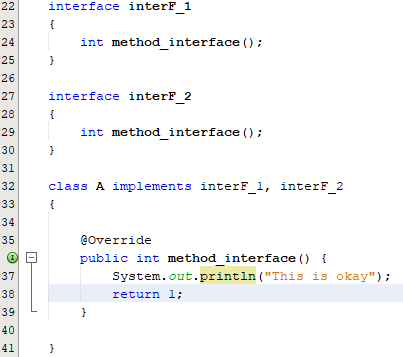
* Method overriding occurs only when two methods name and type signatures exactly match. If they are not, then the two methords are just over loaded.
* Only Non-static, non-final, non-private functions can be overridden (Virtual functions).
* Only instance methods can be overridden. Static methods cannot be overridden (They can be hidden like variables hiding).
* OR - If a subclass defines a static method with the same signature as a static method in the superclass, then the method in the subclass hides the one in the superclass.
* For overriding methods – name and parameters of the method should match. Access modifier (It has to be same or broader) and return type need not match (can be a subclass).
* Java allows for Covariant Return Types, which means you can vary your return type as long you are returning a subclass of your specified return type.
* If super class method throws an exception, the overridden method must throws the same or narrower exception

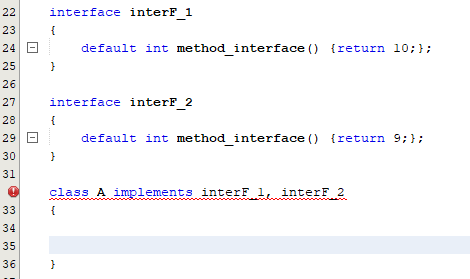
** 🡨 Summary**

**Example (covariant return type)-**

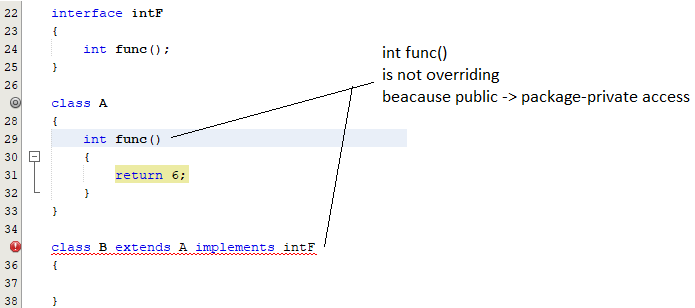


**Example 2 (two interface with same method)–**

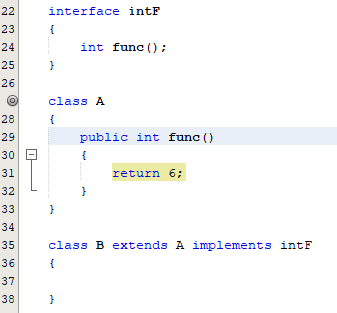
 **🡨 No issues in this code**

 **🡨 Not okay (Class A has to override the default functions in this case, because there are two functions with same name in the interface)**

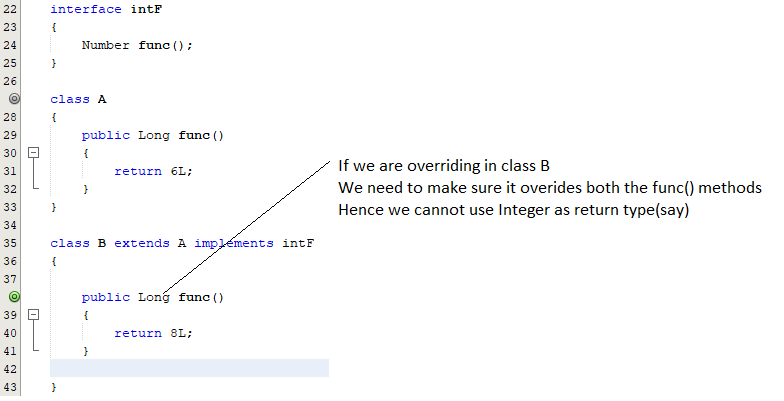
**Example 3 (extended class overriding)-**

****

**Solution to above issue**



**Example 3.1 –**

****

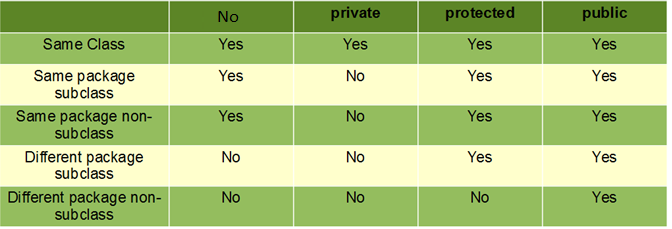
**super and super() –**

* super – access super class members (variables and methods)
* super() – call super class constructor

**abstract class –**

* Abstract class in java can’t be instantiated.
* If a class have abstract methods, then the class should also be abstract using abstract keyword, else it will not compile.
* We can mark a class as abstract even if it doesn’t declare any abstract methods.
* The subclass of abstract class in java must implement all the abstract methods unless the subclass is also an abstract class.
* Java Abstract class can implement interfaces without even providing the implementation of interface methods.
* We can run abstract class in java like any other class if it has main() method.
* Protected modifier can be applied to a abstract method

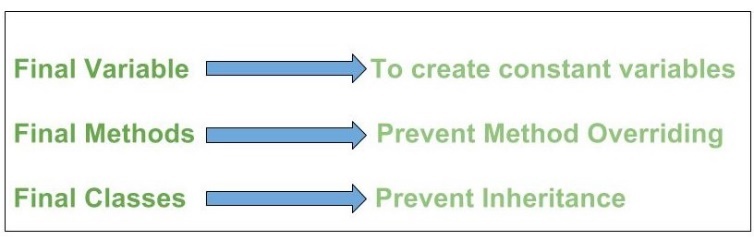
**Access Modifiers –**

****

* Private - Only in the same class
* Public – Everywhere
* Protected – same package || subclass
* no modifier / package-private – same package

**private < package-private < protected < public**

**final –**



* Variable – constant
* Function – Cannot be overridden
* Class – Cannot be inherited

**non-static final variables can be assigned a value -**

either in constructor

or

with the declaration.

**static final variables -**

cannot be assigned value in constructor

they must be assigned a value with their declaration.

**Constructors –**

* Compliler inserts a default no argument constructor only if the class has no other constructor.
* If super class contains only one constructor (not a no argument constructor); Then the child class my make a manual call to the super classe’s constructor in its constructor.

**Method signature –**

1. Return type must always be just before the name of the function.
2. public static / static public - Both are correct
3. final public static / public static final / static final public – all combinations are correct
4. For main function – public, static, void and String are all must
5. Class containing the main() function can be abstract
6. public static void main(String[] args)

public static void main(String[] abc)

static public void main(String[] args)

public static void main(String args[])

public static void main(String...args)

public static void main(final String[] args)

public final static void main(String[] args)

public synchronized static void main(String[] args)

public strictfp static void main(String[] args)

final static synchronized strictfp static void main(String[] args)

1. **Constructors should not have any return type.**
2. **Varags-**

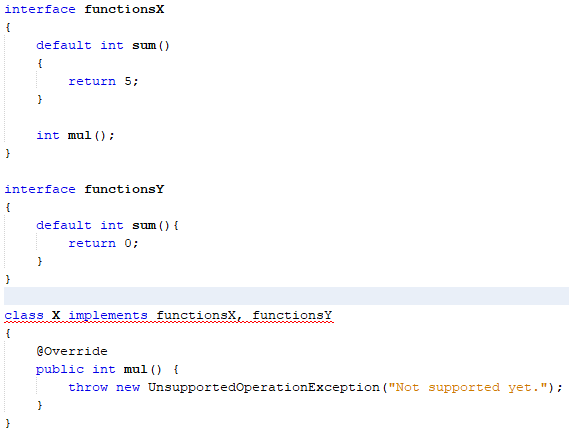
* **Always the last argument**
* **Only one in the function call**

**import static –**

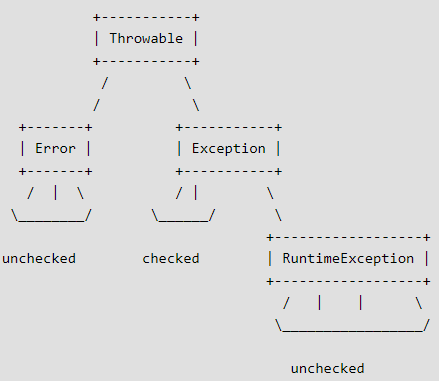
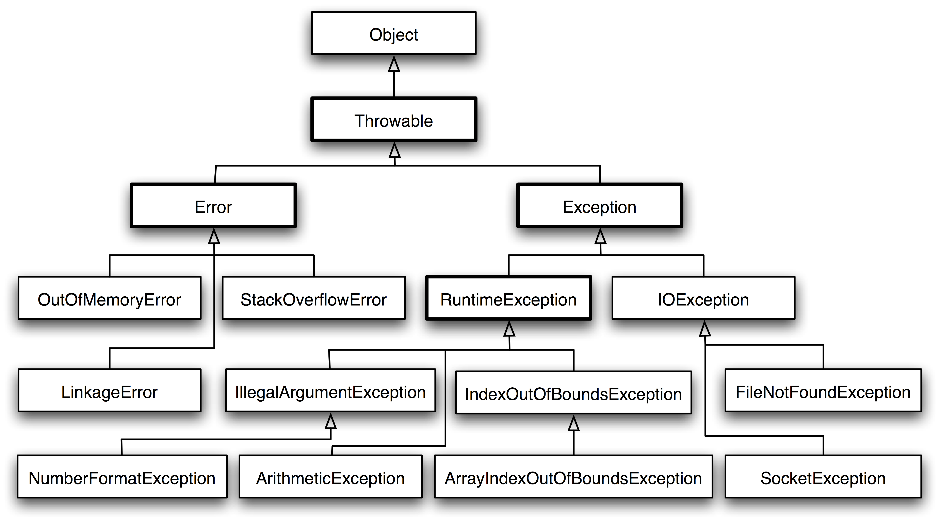
**With the help of static import, we can access the static members of a class directly without class name or any object. For Example: we always use sqrt() method of Math class by using Math class i.e. Math.sqrt(), but by using static import we can access sqrt() method directly.**

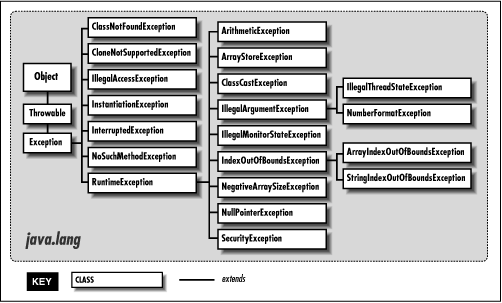
**Interface –**

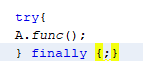
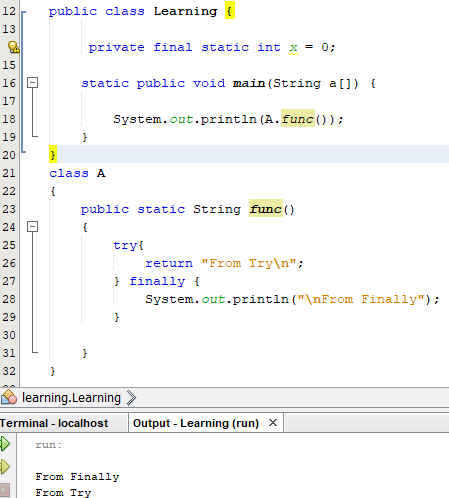
* **One class can implement any number of interfaces.**
* **Each class that implements an interface must implement all its methods.**
* **An interface cannot implement another interface**
* **An interface can extend another interface**
* **From java 8 onwards an interface can contain default and static methods**
* **All overriding methods (overriding interface methods) must be public.**
* **Interface methods cannot be declared as – protected, private, final**
* **Variables – implicitly public, final and static (must be initialized)**
* **Functions – implicitly public (can be declared a static)**
* **If a class implements two or more interfaces which implement identical default methods, it leads to compilation error. To avoid compilation error, the class must override the methods implemented in the interfaces.**

** Class X must override sum() to avoid compilation error**

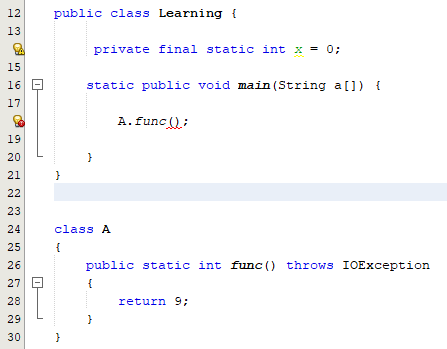
**Exception Handling –**

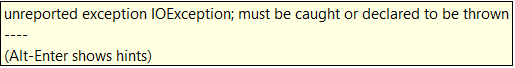




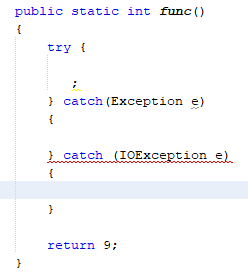
* **Try block must contain a catch or finally block (atleast one among the two)**  🡨works!
* **Try 🡪 catch 🡪 finally (order is important)**
* **It is recommended not to catch Error – Error often indicates failure in JVM**
* **Stack trace is printed when an exception is not caught.**
* **Checked exceptions – Should be handled (or else compiler error)**
* **Unchecked exception – Need not be handled**
* **finally block is executed at any cost (even if there is a return in the try block or if there is a exception thrown in catch block)**
*  **🡨 Finally executes, only then function returns**
* **If both catch and finally throw an exception,**

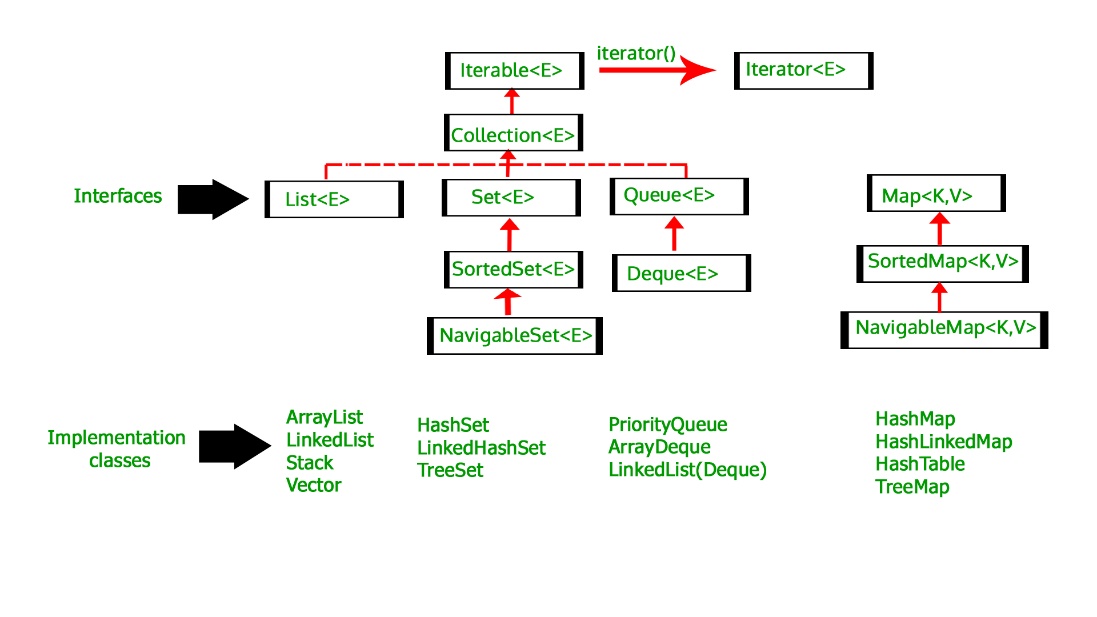
**Then exception from catch block is dropped and only exception of finally block is sent back to caller.**

 **🡨 IOException is a checked exception**

****

**🡨 Null pointer exception is unchecked**

 **🡨 Cannot reach IOException**



**/\***

**\*\* Thread class - java.lang.Thread**

**\*/**

**//Extending thread**

|  |  |  |
| --- | --- | --- |
| public class Global\_Data {    public static int money = 0;    } | public class Increment\_Money **extends Thread**{    Increment\_Money()  {  }    public void run()  {  while(true)  Global\_Data.money += 1;  }    } | public class LearningJava\_2 {  public static void main(String[] args) throws InterruptedException {  **Increment\_Money MI = new Increment\_Money();**  **MI.start();**  while(true)  {  System.out.println(Global\_Data.money);  Thread.sleep(10);  } } }  }    }    } |

**//Implementing Runnable**

|  |  |
| --- | --- |
| public class Decrement\_Money **implements Runnable**{  public void run()  {  while(true)  Global\_Data.money += 1;  }  } | public class LearningJava\_2 {  public static void main(String[] args) throws InterruptedException {  Increment\_Money MI = new Increment\_Money();  MI.start();    **Decrement\_Money DI = new Decrement\_Money();**  **Thread DI\_T = new Thread(DI);**  **DI\_T.start();**    while(true)  {  System.out.println(Global\_Data.money);  Thread.sleep(10);  }    }    } |

|  |  |
| --- | --- |
| **//Using Lambda expression**  Runnable DI = () ->  {  Global\_Data.money -= 1;  };    Thread DI\_T = new Thread(DI);  DI\_T.start(); | **//Using Inner class**  class dec implements Runnable  {  public void run()  {  Global\_Data.money -= 1;  }  }  dec DI = new dec();  Thread DI\_T = new Thread(DI);  DI\_T.start(); |

**/\***

**\*\* Scanner class - java.util.Scanner**

**\*/**

**// Scanner Constructors**

Scanner(File source)

Scanner(InputStream source)

Scanner(Path source)

Scanner(Readable source)

**// From standard input**

Scanner scn = new Scanner(System.in);

**// Scanner from file**

FileReader fr = new FileReader("E:\\NewFileForJava.txt");

Scanner sc = new Scanner(fr);

**/\***

**\*\* Regular expression - java.util.regex.Matcher java.util.regex.Pattern**

**\*/**

Pattern pat = Pattern.compile("\\d+ABC\\d{2}");

Matcher matr = pat.matcher("This is the string that will be searched 1345145ABC455234GHII");

if(matr.find()) // 🡨 Find first to avoid illegal state exception

System.out.println(matr.group()); // output --- 1345145ABC45

String replaced\_String = matr.replaceAll("Guru Sarath");

**/\***

**\*\* String tokenizer - java.util.StringTokenizer**

**\*/**

StringTokenizer st = new StringTokenizer("A.X.G", ".");

while(st.hasMoreTokens())

System.out.println(st.nextToken());

**/\***

**\*\* Random - java.util.Random**

**\*/**

Random r = new Random(23);

r.nextInt();

**/\***

**\*\* File - java.io.File**

**\*/**

File f1 = new File("E:\\NewFileForJava.txt");

f1.getName();

f1.getPath();

f1.isFile();

f1.lastModified();

f1.exists();

f1.setWritable(true);

File f2 = new File("E:\\Guru.txt");

f2.createNewFile();

File d1 = new File("E:\\NewFolder\_JAVA");

d1.mkdir();

File dir = new File("E:\\");

String listOfDirs[] = dir.list();

File fileList[] = dir.listFiles();

for (String d : listOfDirs)

System.out.println(d);

**/\***

**\*\* URL - java.net.URL / java.net.HttpURLConnection / java.net.MalformedURLException**

**\*/**

try{

**// Build URL**

URL urlX = new URL("https", "docs.oracle.com", "/javase/7/docs/api/java/net/URL.html");

} catch (MalformedURLException e)

{

e.printStackTrace();

}

System.out.println(urlX.toString());

HttpURLConnection connection\_oracle = (HttpURLConnection)urlX.openConnection();

InputStream ips\_oracle = connection\_oracle.getInputStream();

Scanner scn = new Scanner(ips\_oracle);

scn.useDelimiter("\\A");

while(scn.hasNext())

{

System.out.print(scn.next());

}

**/\***

**\*\* JSONObject - org.json.simple.JSONObject**

**\*/**

