Date

Working Of Garbage Collection.

1) Garbage Collection is nothing but execution of System.gc() statement.

2) This statement can be called by either programmer or jvm but decision to perform garbage collection is taken only by JVM

3) This means that although the programmer would write System.gc() in his program then is no guarantee that JVM would accept the request.

Ex:

```
class A
{
  psvm(-)
  {
    A a = new A();
    A a1 = new A();
    a = null;
    System.gc();
  }
}
```

Q: How many objects are present in heap.

Ans: Cannot be determined.

#_____

#_____

Date

Packages & Access Specifiers.

Rules:

1) There can be only one public class per source code file.

2) 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 A { } must be in a source code file named A.java.

3) If the class is a part of package, the package statement must be the first line in the source code file, before any import statement that may be present.

4) If there are import statements, they must go between the package statement (if there is one) and the class declaration.

Date

5) A file can have more than one non public class.

Access Specifiers (Visibility Labels)

Def:

- Java Access Specifiers (also known as Visi--- ~~labels~~ Specifiers) regulates access to classes, field and method in Java. These Specifiers determine whether a field or method in a class, can be used or invoked by another method in another class or sub-class. Access Specifiers can be used to restrict access. Access Specifiers are an integral part of object-oriented programing

#

Types of Access Specifiers.

Public
default
protected
Private;

Summary of access specifiers.

Access Modifiers	Def	Pri	Prot	Public
— Accessible inside the class	Y	Y	Y	Y
— Accessible within subclass inside same package.	Y	N	Y	Y
— Accessible outside the package	N	N	N	Y
— Accessible within subclass outside same package	N	N	Y	Y

#

#

Date

Primitive Data Type and Wrapper Classes.

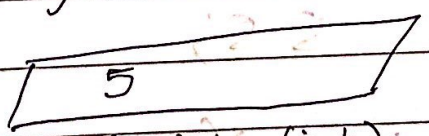
Following data types are available in Java.

1) ~~Byte~~ byte (8 bits)
2) Short (16 bits)
3) char (16 bits)
4) int (32 bits)
5) float (32 bits)
6) long (64 bits)
7) double (64 bits)
8) boolean (true, false)

Ex: 1

```
int x = 5;
```

memory



32 bits (int)

range : -2^{16} to $+2^{16}$

Note:

$2^{16} = 65000$

$2^{32} = \ldots\ldots$

What makes you happy?
#HappyCollegeDays

ex 2:

byte b = 120; ✓

byte b1 = 130; ✗

$2^8 = 256$

range: −128 to +127

Eg:

① long l = 5l;

int x = 6;

l = x; ✓

x = l; ✗

② float f = 3.2F;

Double d = 5.5;

f = d; ✗

d = f; ✓

Date

(3) int x = 5;
float y = 5.5f;
y = x;

(4) float f = 3.2f;
long l = f;

Note:

JVM Says

Date

Note

In eg(1) as container size of x(int) is less than long and we are not losing out of any information it is allowed.

In eg(2) of f = d means trying to fit a double inside a float which is not possible.

In eg(3) an int is assigned to float which means we gained the precision and so its fine.

In eg(4) although float can fit inside long we lose the precision and hence its not allowed.

Date

Primitive Casting

Eg:

① ~~long l = 5L;~~

~~int x = 6;~~

~~l = x; ✓~~

① long l = 6; ✓

② long l = 6.0; ✗ (Error: Possible loss of precision).

long l = (long) 6.0; ✓

sop(l); o/p → 6

③ float a = 5;

sop(a); // 5.0 ✓

④ double d = 5L;

sop(d); // 5.0 ✓

⑤ double d = 5.5f; ✓
sop(d); // 5.5 ✓

⑥ long l = (int) 5.5; ✓
(long) 5.5; ✓
sop(l); // 5

⑦ char c = 'a'; ✓
char c = 100; ✓
sop(c); // d

char x = -100; X
char x = (char) -100;
char y = -70000;
char y = (char) -70000;

Date

~~Whitening~~

Widening and boxing in Java.

Priority
1. Widening
2. Boxing
3. Var-Args.

Widening:

Ex: ① double d = 5.5F;

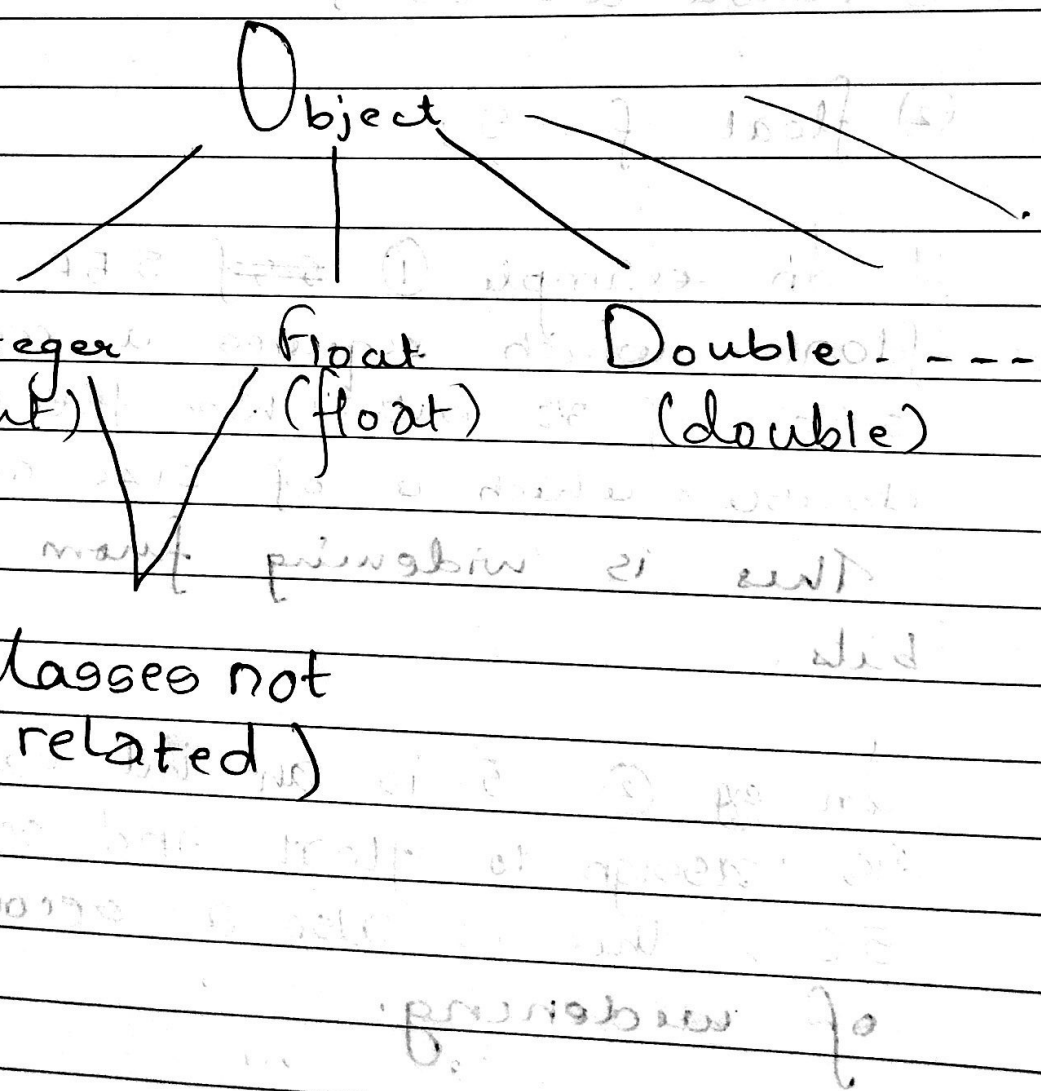
② float f = 5;

In example ① ~~5.5~~f 5.5F is a float which requires a container size of 32 bits; We fit this into double which is of size 64 bits.
This is widening from 32 to 64 bits.

In eg ② 5 is an int value which is assign to float and so becomes 5.0, this is also a second type of widening.

Boxing (Auto)

In Java, int is boxed to Integer, float is boxed to Float and likewise all primitive variables are boxed to respective classes.



Date

Variable Arguments (Var-args)

Ex:

① class A
{
void m1(int... x)
{
sout("int");
}
psvm(-)
{
new A().m1(5);
new A().m1(5,5);
new A().m1(5,5,5,5);
}
}

Rules:

a) Int and Integer are similar in variable arguments and hence void m1(Integer... z) { } and void m1(int... y) { } are together not allowed.

(1) This also applies to float, Float, Double, double and so on for all primitive types.

(2) Variable argument should always be last in the seq.. of argument.

Ex: void m1(int x, float ... y){} ✓

last in Seq..

void m1(float ... x, int y){} ✗

Not allowed

(3) As variable args.. should be last in seq, we cannot write multiple arguments, because both can't be last!

Date

Example:

```
class A
{
  void m1 (int ... x)   // var-args
  {
    sout ("int ... ");
  }
  void m1 (Integer ... x) // Boxing
  {
    sout ("Integer");
  }

  void m1 (float x)  // Widening
  {
    sout ("float");
  }

  psvm(-)
  {
    new A().m1(5);
  }
```

O/p float.

Date

	std's	calls	o/p
①	m1(int.. x) m1(object x)	m1(5)	m2(object x)
②	m1(int x) m1(byte b) m1(Short s)	short s=5; m1(5); = exact match m1(s); = int (widening	
③	m1(float x) m1(long x) m1(double x)	m1(5); priority: long float double	m1(long x),
④	m1(long x) m1(double x)	m1(5.5F)	m1(doub x)

What makes you happy?
#HappyCollegeDays

#

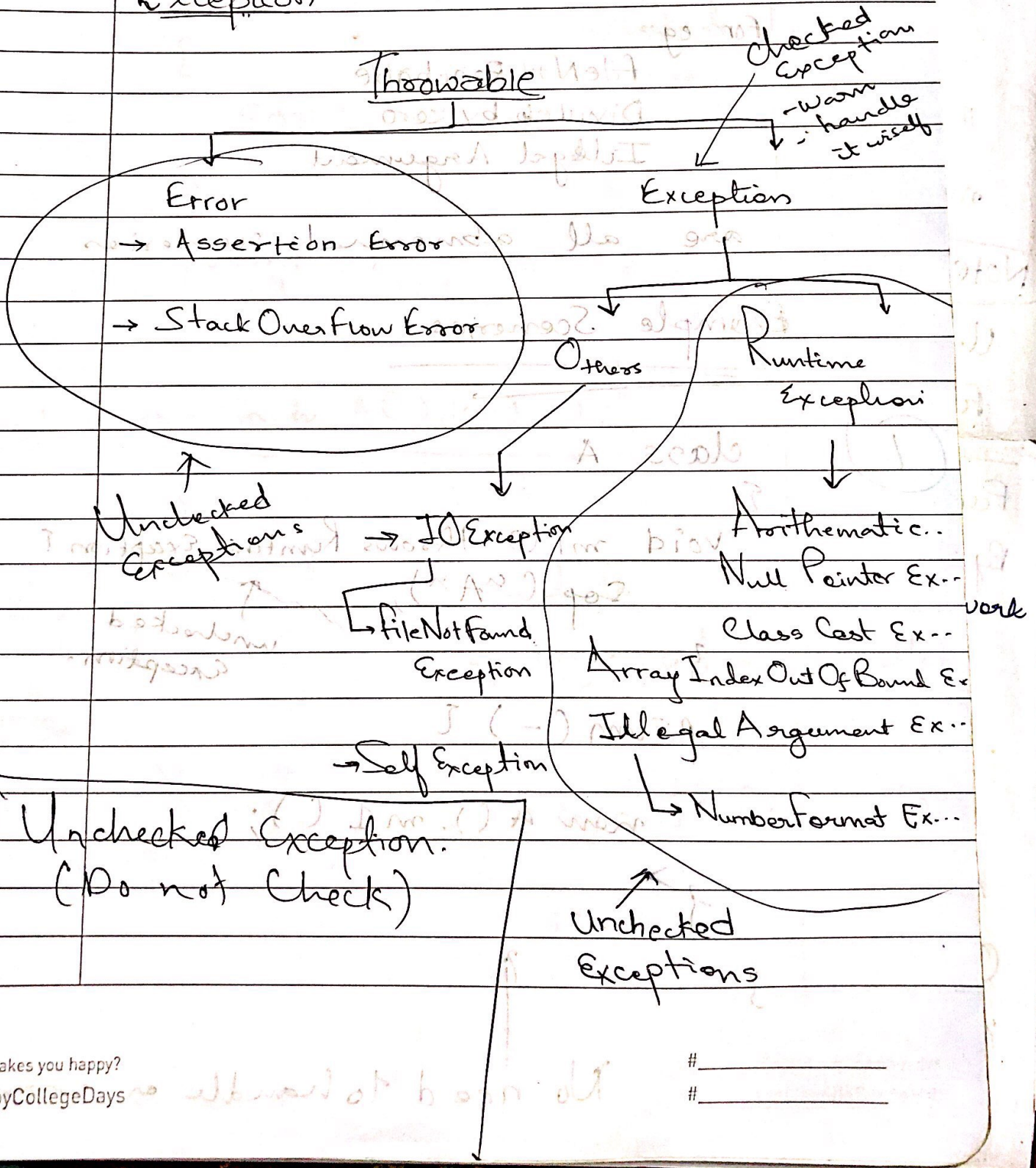
#

Date

Exception Handeling :

Why exceptions ?

Exception



Exceptions are used in Java to warn or handle abnormal scenerios.

For eg:
- FileNotReachable
- Divided by zero
- Illegal Argument

are all abnormal sceneriors.

Example Scenerios

(1)

```
class A
{
void m1() throws RuntimeException {
    sop("A");
}
psvm(-) {
    new A().m1();
}
}
```

↑ unchecked exception.

↑ No need to handle or warn.

Date

② class A {

void m1() throws IOException
↑ checked

{

Sop("A");

}

psvm(-)

{

new A().m1();

}

}

Need to

① try-catch (handle)

or

② throws (warn)

What makes you happy?

#

#

Try - Catch

Rules

1) try { }
catch (Exception e) { }
finally { }

valid

2) try { }
finally { }

valid

(3) try
{
}
catch (Exception e)
{
}

valid

4) try { }
try { }
catch (Exception e) { }

Not Valid

becoz 1 try - 1 catch.

What makes you happy?
#HappyCollegeDays

#

#

Date

Note. 1

There are 3 ways of throwing an exception.

i) Write code that throws an exception

```
int a=5;
int b=0;
int c= a/b;
```

ii) Throw new Arithematic Exception();
↳ Manual throw.

iii) Arithmetic Exception e = new Arithmetic Exception();

throw e;
↳ Manual throw.

Note 2.

To throw a checked exception manually, we have to warn the JVM first, however to throw unchecked exception there is no need to warn.

Ex:

(1) psvm(-) throws Exception
{
try
{
throw new exception();
}
finally { }
}

✓ (Manual Throw)

(2) psvm (-)
{
try {
throw new NullPointerExcptio.();
}
}

Unchecked Exc. ~

What makes you happy?
#HappyCollegeDays

#

#

Example:

(1)

```
class A {
    psvm (-) {
        try {
            sout ("Welcome");
            throw new ArithmeticException();
                //Unchecked So no need to
        }                   warn or catch.
        catch (ArithmeticException e) {
            Sout ("Arithemetic Problem");
        }
        finally {
            Sout ("Finally");
        }
    }
```

Output:

Welcome
Arithetic Prob.
Finally

Note:

If an unchecked exception is thrown and not caught, we get a red message in the output. (Exception in thread main)

② class A {

psvm(-)

{ try

{

sout ("Welcome");

~~throw new Except~~

throw new IOException();

// checked, so either warn or catch or both.

}

catch (IOException e) { // catching

sout ("Arithmetic Exception");

}

finally { sout ("Finally"); }

}

Date

Note:

If a checked Exception is throw, We have to either catch it, or we can do both.

Multiple- Catch Blocks

Rules:

1) In multiple catch blocks JVM selects most relevant first.

2) We have to follow the hierarchy (lower to higher) (means lower first)

Example.

```
try
{
throw new ArithmeticException();
}
catch (ArithmeticException e)
{
    sop("Arithmetic");
}
catch (RuntimeException e)
{
}
catch (Exception e)
{
}
```

What makes you happy?
#HappyCollegeDays

* ~~Exe~~ Exception in Overrided Methods.

3 scenerios

(1) void m1() throws Exception.
void m1() throws _____

1) All Unchecked
2) Exception
3) IO Exception
4) All checked (except throwable)

(2) void m1() throws FileNotFound
void m1() throws _____

1) All Unchecked
2) File Not found.

(3) Void m1() throws IOException, Error.
void m1() throws _____

1) All Unchecked
2) IO Exception.
3) FileNot found.

What makes you happy?
HappyCollegeDays

#_____
#_____

* Self Exception:

1) To create our own exception class, it must extend generic exception class.

Ex:

```
class MyException extends Exception { }
```

2) This My exception goes in the others section of the hierarchy.

* Nested Try-Catch Exception:

Exam Watch:

1)

```
try
{
    throw new ArithmeticException();
}
catch (Exception e)
{
    throw e;
}
```

AE

Ex

Ex e = new AE();

this throws AE.

2) i) class A
ii) {
iii) psvm(-)
{
iv) ArithmeticException e = null;
try
{
throw e; // Null pointer exception.
}

```
catch (Exception e1)
{
    e1 = e;

}

}
```

In the above example is reference e at line 4 is a local variable and should be initialized to null.

As it is declared in the main fn(), the scope remains throughout the main and hence it should not be reused anywhere.

If e is as written at line 2 outside the main fn() it becomes instance variable which by default is initialized so we do not have to initialize it.

Date

If a variable is declared inside a try block, its scope remains inside a try block and can be reused outside try block.

Same rule applies to catch and finally block as well.

3) If a exception is thrown using a code, we can write statements after it, however if an exception is thrown manually, anything written after that is not reachable and generates compilation failure.

Ex. class A {
psvm(-){
try {
throw new AE();
sop("—");
}
catch(Ex e) { sop("catch"); }
}

int n = 10/0;

EW

4) Method Propagation.

```
class A
{
public void m1() throws Exception {
    m2();
}

private void m2() throws Exception {
    m3();
}

private void m3() throws Exception {
    throw new IOException();
}
}
```

Date

```
class B extends A
{
psvm(-)
{
  try{
      new B().m1();
    }

  catch(Ex:eLq)
  {
    Sop ("Exception1");
  }
```

Output

Exception1

Assertion's :=

Assertion's are used to ensure that the code does not execute ~~unwa~~ or process unwanted conditions.

Syntax:

```
assert exp1 : exp2;
```

~~Cond~~

where

i) exp 1 should return a boolean value.

ii) exp 2 can be a string, arithmetic operation or just a variable.

iii) exp 2 is optional and so can be skipped.

iv) if we use mtd's as exp 1, then it must return a boolean value.

If we use mtd as exp 2, then it must return something preferably string, and cannot be void.

Date

Note:

Assertion by default are disable by JVM, to enable them we write

java -ea <class name>

↓

enable assertion

at runtime.

Base Rule:

i) If exp 1 (evaluates to true, we ignore the complete statement.

2) If exp 1 evaluates to false, we throw a red color assertion error., with display of exp2. [if available]

3) When the assertion is evaluate to false, program does not procede further.

#

```
package ocpjp;

class A {
    int n = 1;
    void m1 (int marks)
    {
        assert m1() : m2() ;
        System.out.pln ("Marks are" +
                                marks);
    }

    public static void main(String[] args)
    {
        new A .m1(120);
    }
    boolean m1 ()
    {
        return false;
    }

    A m2()
    {
        return new A ();
    }
}
```

O/p : Assertion error.

Date

-) assert false;
assert true : 1;
assert x == 10;
assert str. = str;

Date

Guidelines for Using Assertion:

1) Assertions should not be used to check the argument of mtd which is public, instead use illegal argument exception.

2) In Assert statement, do not change the value of a variable.

Eg: assert ++x > 0; ✗
assert x > 0 : x++; ✗

Handling Assertion Error

1) We can handle assertion error using ~~using~~ a try catch B block work using fall
   i) Assertion error.
   2) Error.
   3) Throwable.

Note: fall.. assertion syntax are legal and can be used.

What makes you happy?
#HappyCollegeDays

Inner Classes

Date

There are four types of inner classes.

i) A class inside a class.

Eg:

```
class A
{
  int x;
  void display()
  {
    sout("A");
  }
  class B
  {
    int x;
    void display()
    {
      sout("B");
    }
  }
psvm(-)
{
  A a = new A();
  A.B b = a.new B(); // A.B b = new A().new B();
}
```

Copy

O/P: B

What makes you happy?
#HappyCollegeDays

#

#

(ii) Static Inner Class.

{

Paste

}

```
psvm(-)
{
    A.B b = new A.B();
    b.display();
}
```

O/p : B.

(iii) Method Local Inner Class.
(A class created inside the method).

Ex → PTO

```
class A
{
 int x;
 void display()
 {
  class B
  {
   ~~int x;~~ final int y = 5 //must be declared as final.
   void display()
   {
    sout("B");//sout(" y);
   }
  }
   B b = new B();
    b.display();

}

psvm(-)
{
 new A().display();

}

}
```

O/P : B

#_____

#_____

Date

Note: To use a variable declared inside the mtd in the class, we have to declare it as final.

Date

Multi-Threading :

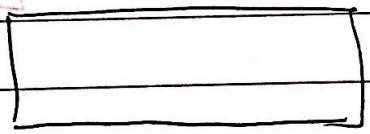
A program by default always run's in main thread.

A programmer can start a new thread manually and perform its own execution independent of main thread.

Consider fou.. scenerio.

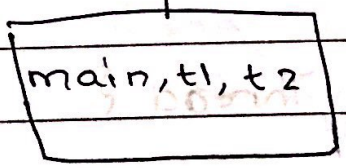
Threads

main
T1
T2



Any one at a time.

Thread Scheduler

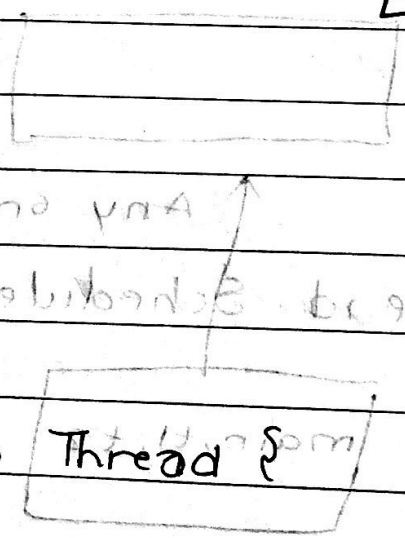


Syntaxes for Creating and Starting Thread.

→ If a class wants to participate in multithreading, it has 2 choices.

i) extends ~~thread~~. → (Class)

ii) implements Runnable → (Interface)



Note :→

```
class Thread {

    public void start()
    {
        run();
    }
```

Date

```
public void run()
 {

 }
}
```

```
interface Runnable {

    void run();

}
```

To start a thread we have to call the start() method of the thread class. There is no otherway of starting a thread.

Scanned with CamScanner

Type 1:

```
class A extends Thread
{
  public void run() // Overridding run.
  {
   sout (Thread.currentThread().getName());
  }


public static void main(-)
{
  new A.start(); // ~~start~~ New thread
                    Starts

  new A.run(); // main thread.
  }

}
```

By default JVM assigns name to the thread. We can assign our name to thread by calling the constructor of thread class and passing the name as string.

Type-2

Using an object of Thread class.

```
class A extends Thread {
    public void run() {
        Sout (Thread.currentThread().getName());
    }

psvm(-) throws InterruptedException
{
    Thread t1 = new Thread(new A(), "A");
    Thread t2 = new Thread(new A(), "B");

    t1.start();
    t2.start();

    t1.join(); //main will get executed after t1
    t2.join(); //main will get executed after t2

    Sout("Thank");
}
}
```

```
A (String name)
  {
     super (name);
  }

psvm (-)
  {
   new A ("A") . start ();
   new A ("B") . run ();
    }
}
```

O/P A OR main
main A

Date

Type 3:

Implements Runnable.

```
class A implements Runnable
{
    public void run()
    {
        sout (Thread.currentThread().getName());
    }
    public stat
    psvm(-)
    {
        Thread t1 = new Thread (new A(), "A");
        t1.start();
    }
}
```

Type 4:

Anonymous [redacted] Day.

```
class A {

pu psvm(-) throws InterruptedException
{
  Thread t1 = new Thread("A")
  {
    public void run()
    {
      sout(Thread.currentThread().
                    getName());
    }
  };

Thread t2 = new Thread("B")
 {
   public void run()
  {
    sout(Thread.currentThread().
                    getName());
  }
 };
  t1.start();
  t2.start();
 }
}
```

Anon

#_____
#_____

Type 5.

Anonymous using Runnable.

```
class A
{
  psvm(-) throw InterruptedException
  {
    Runnable r = new Runnable()
    {
      public void run()
      {
        Sout(Thread.currentThread().
                                getName());
      }
    };

    Thread t1 = new Thread(r);

    t1.start();

  }
}
```

Type 6 :

```
class A {
    psvm(-) throws InterruptedException
    {
    new Thread(new Runnable()
        { public void run()
            {. }.
    }
                ).start();
    }
}
```

Thread Synchronization

There are 2- ways in which (synchronization) can be implemented.

1) → Synchronized Block [needs an object]

2) → synchronized Method.

Synchronized Blocks

```
class A extends Thread
 {
  String str = " ";
  public void run ()
  {
   synchronized (str) // takes an object.
   {
    sout(Thread.currentThread().getName());
    sout ( ——————— u ——————— );
   }
  }
```


#_____

#_____

Date

psvm(-) throws InterruptedException

{

Thread t1 = new Thread(new A(), "A");

Thread t2 = ——v—— (new A(), "B");

t1.start();

t2.start();

}

}

2) Synchronized Method.

```
Class A extends Thread
{
    public void run()
    { m1(); }

Static synchronized void m1 ()
{
 sout (Thread currentThread().getName());
 try {
        sleep (1000);
      }
 catch (Int ex)
 { }

Sout (Thread.currentThread().getName());

}
```

Date

```
public static void main (—) throw
                                Interrupt
Thread t1 = new Thread (new A(),
                        "A");
Thread t2 = new Thread (new A(),
                        "B");
    t1.start();
    t2.start();
}

}
```