Date - 01-08-2022
Monday

Instructor - Manju

Introduction to Java

What is programming language?

- set of instructions for computer to follow
- Examples of programming languages:
  Java, Python, C++, C and many more

History of Java Programming

↳ James Gosling (Father of Java)
↳ invented java in early 90s.
↳ overcome disadvantages of C & C++

Green Talk → Oak → Java (1995) → name of coffee shop

Features of Java Programming

> Highly secure language → Banking, Server Maintainence
> Platform Independent

C/C++	Java	
made in windows → run on → windows ✓, linux ✗, mac ✗	made in windows → run on → windows ✓, linux ✓, mac ✓	> Object Oriented Prog. lang. (object, class)
Platform dependent		> Easiest language (Syntax)
		> Automatic Garbage Collection — unused memory is automatically collected

Bansal

Use cases of Java Programming

(i) Web Application – Browsers (Chrome, Opera)
(ii) Business App. – Trading
(iii) Mobile Application – Spotify

5 apps → 2 (fully based on Java)
→ 2 (Java for maintenance)
→ 1 (partially needs Java)

(iv) Gaming Application (multi-threading needed)
(v) Cloud Based Application
(vi) Desktop GUI

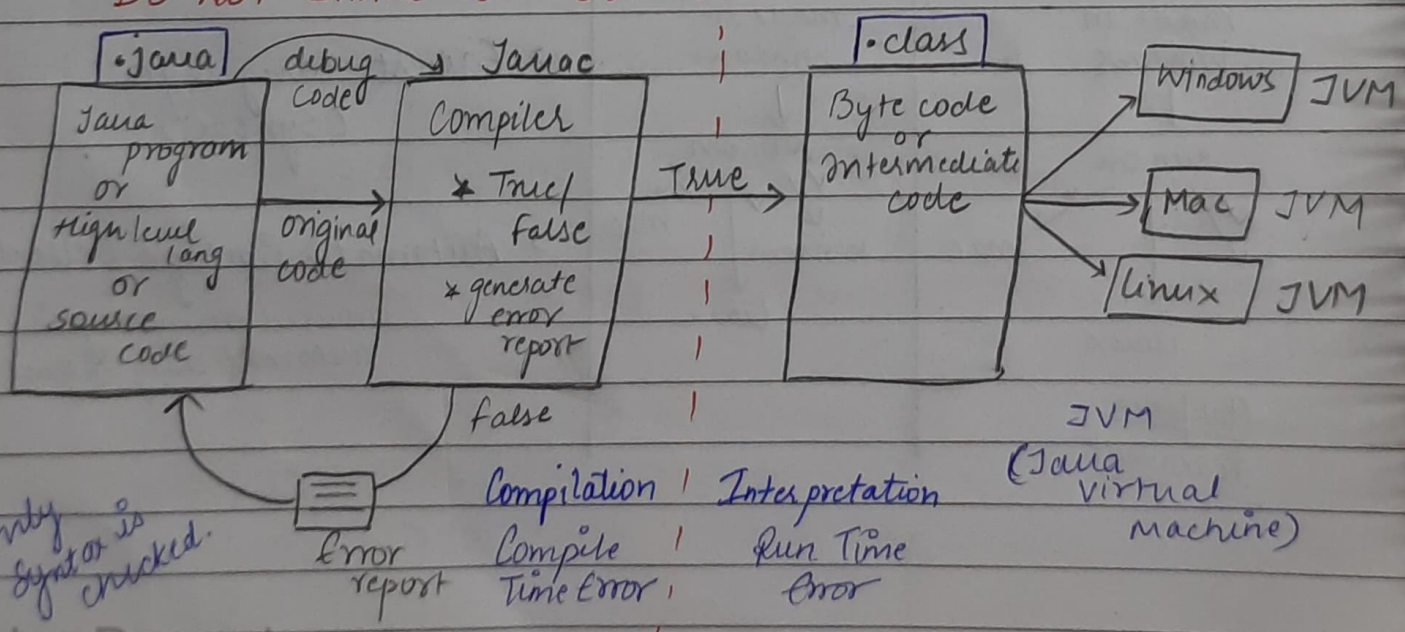
What is Java?

- programming language
- high level language
  (Human understandable language)

How does a Java Program works?

* Java program is also known as source code and high level language (.Java extension)

DO NOT SHARE YOUR SOURCE CODE



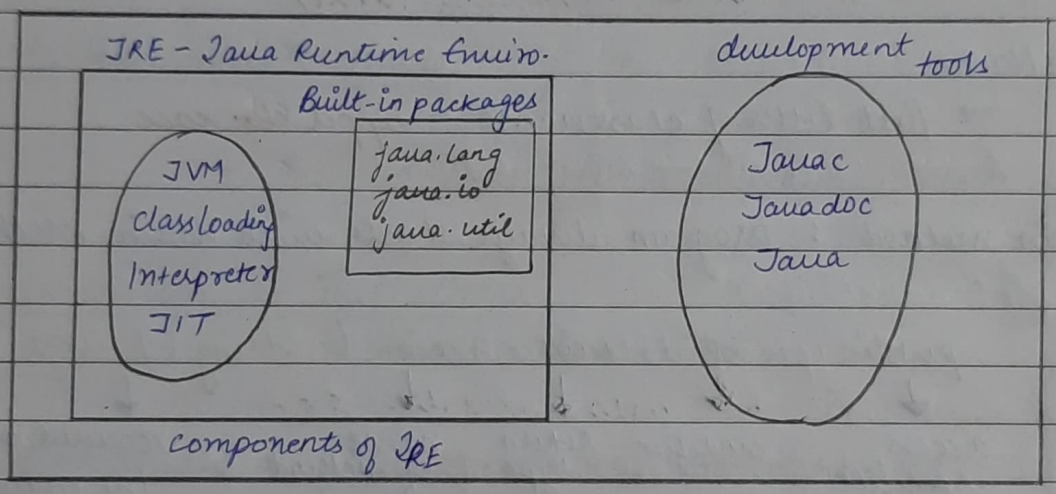
Isha Bansal

DATE
PAGE

JVM - Java Virtual Machine

- Reads byte code line by line and converts into machine level language (Binary language)
  ↓
  machine understandable language
- plat-form dependent
- present in JRE (Java Runtime environment)

JDK - Java Development kit



(i) JRE (Java Runtime Environment) - needed for execution of java program.

(ii) JVM is a scheduler
↳ calls interpreter for conversion of byte code → ~~machine code~~

(iii) JIT
↳ Just in Time
↳ improves efficiency & time management

WORA ⇒ Write One Read anywhere

sha Bansal

DATE
PAGE

Structure of Java Program

- Java program is always written inside the class
- Create a class → create a block for the class

class { → block indication
}

College has many classrooms
↳ to identify our department we require the 'name' of the department.

- className
  ↳ first letter of class name - upper ~~cla~~ case

- main method : program always starts with main method

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

public	static	void	main	(String [] args)
access specifier	modifier	return type	name of method	required by the main method

String : 'S' is in capital letters

```
class ClassName {
    public static void main (String[] args) {
        // set of instructions
    }
}
```

Class block
main block

Members of Classblock
- Initializers
- variable
- methods

Isha Bansal

DATE
PAGE

Introduction to Variables

- containes used to store & update values
- can store anything

variable ← [20] → a [John] → name variable

declaring a variable

<datatype> <variable name> = <variable value>

Introduction to Methods

- Functions are renamed to Methods
- Block of instructions to perform some tasks

Introduction to Initializers

- execute the startup instruction

What is the program doesn't have main method?

- compile time successful (.class file ✓)
- run time error occurs

PRINTING STATEMENT IN JAVA

#1 System.out.println();	#2 System.out.print();
Prints in a new line	Prints in the same line
print → print the statement; ln → add a new line	print → print the statement

Isha Bansal

Working with Eclipse

> File → New → Project
> Project Name
> Don't create > No

] Create Project

> Ctrl + N → Packages
> Enter Name → Enter

] create Package

> Ctrl + N → Class (Java Class)
> Enter Name → Enter

] create Class

X — Code ① — X

Data Types Introduction

- Types of data we have

DATA TYPE
- Primitive
- Non-primitive

Numbers
- Integer
  - byte 1 byte = 8 bits → 0 (default value)
  - short 2 byte = 16 bits → 0 (default)
  - int 4 byte = 32 bits → 0
  - long 8 byte = 64 bits → 0
- Float
  - float 4 byte = 32 bits → 0.0f
  - double 8 byte = 64 bits → 0.0

Character → char 2 byte = 16 bits → null

Boolean → boolean (True / False) 1 byte = 8 bits → false

Isha Bansal

DATE
PAGE

Primitive datatypes < Number, Character, Boolean

primitive values are stored inside primitive datatypes

Non-primitive datatype
- String (collection of characters)
  String s = "Clarivate";
- All classnames are non-primitive

Variable
- → primitive variable stores primitive data
- → non-primitive variable stores non-primitive data

Introduction to Tokens

Smallest unit of programming language

Types of Tokens
- > Keywords
- > Identifiers
- > Literals

Introduction to Keywords

- ↳ Pre-defined words
- ↳ Programmer can not change
- ↳ 50+ keywords in Java

All keywords should be in lowercase

Introduction to Identifiers

↳ The name given to the components.

Rules of Identifiers
- → can't start with a number
- → '$' and '_' allowed
- → spaces are not allowed

Bansal

DATE
PAGE

Name Conventions

- rules of naming
- compiler doesn't validate conventions
- highly recommended to use conventions

Conventions for classNames

SINGLE WORD : Class (uppercase)

MULTIPLE WORD : ClassName (uppercase)

ClassNameConvention — upper case

Conventions for Variables

SINGLE WORD : var (lowercase)

MULTIPLE WORD : varName (lowercase, uppercase)

varNameConvention — lower case, uppercase

Introduction to literals

primitive values
values assigned to primitive variables

non-primitive values
values assigned to non-primitive variables

Scope of variable

- visibility of a variable

> local
> static
> non-static

x — Code (11) — x

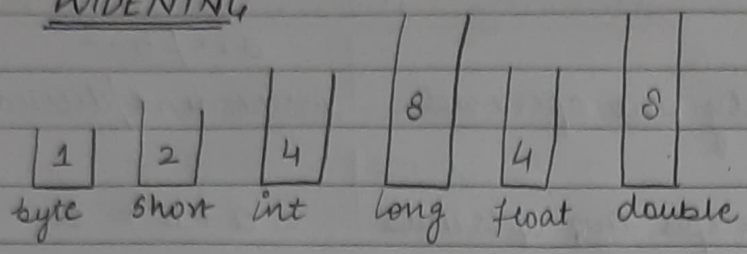
Type Casting

converting one type of data into another type

↳ primitive Typecasting → widening, Narrowing
↳ non primitive Typecasting
↳ upcasting
↳ downcasting
] requires Inheritance

Isha Bansal

WIDENING



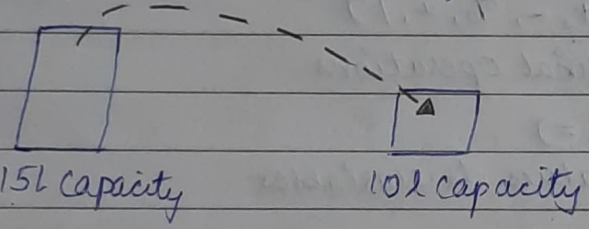
- automatically
- implicity

double d = 9.9f
~~float~~ $\boxed{d = 9.8999}$

int a = 20;
float f = a; $\boxed{f = 20.0}$

NARROWING

15 l of water 10l of water + 5 l waste



* data loss can happen

double d = 75;
float f = d; //error

need typecasting operator

Typecast Operator

double d = 75;
float f = (float)d;

x — Code (III) — x

Introduction to operators

* pre-defined symbols to do specific task

operand

10 + 20 = 30 ⇒ Expression

operators

Bansal

DATE
PAGE

Types of operators

# Unary operator	Only one operand	a++ / b-- increment/decrement
# Binary Operator	accepts two operands	a + b = c
# Ternary Operator	accepts three operands	Conditional operand

Classification of operators

Arithmetic Operator (+, -, %, *, /)
Mathematical operations

Assignment operator (=)
assign values to variables

Relational Operator (==, !=, >, <, >=, <=)
returns only boolean value
compares two values

Increment/decrement Operator
Unary operator, update value of a variable by 1

Conditional operator (condⁿ ? if true : if false)
Ternary operator., return type ⇒ return type of o/p

Logical Operator (&& (AND) || (OR) ! (NOT))
boolean results
comparison

Bitwise Operator

Miscellaneous Operator

x —— code (IV) —— x

Isha Bansal

DATE
PAGE

Increment/Decrement Operator

* pre-increment (++a)
* post-increment (a++)

* pre-decrement (++a)
* post-decrement (a--)

Pre-incr	Post incr	Pre-dec.	Post-dec
int a=5;	int a=5;	int a=5;	int a=5;
int b=++a;	int b=a++;	int b=--a;	int b=a--;
1+5	5+1		
5 6	5 6	5 4	5 4
a b	a b	a b	a b
6	5	4	5
first increment 'a' then store in 'b'	First assign to 'b' then increment 'a'		

int x = 3
int y = x++ + ++x
(3+1) + (1+4)
3 5

x: 3 4 5
y: 8

Decision Making Statements

→ if
→ if-else
→ if-else-if ladder
→ switch

# if statement
if (condition) {
//Set of instruction }

# if-else statement
if (cond.) { }
else { }

# if-else-if statement
if (cond.) { }
else if (cond.) { }
else { }

Ria Bansal

Switch

- Branching statements
- Grouping

```
switch ( variable / value ) {
        case (values/expression) : { }
                                   break;
        :
        case (value) :   { }  break;

        default : {
                                }
}
```

cases

when none of the condition is getting true.

Switch ⇒

* long, double, boolean are not passed

case ⇒

* variables are not allowed

Grouping

```
Case 1:
Case 2:
Case 3:
Case 4:
  { // print statement }
  break;
```

combine action for all 4 cases.

x —— Code (V) & (VI) —— x

Isha Bansal

DATE
PAGE

Looping Statements

- Repeatation
  - for loop
  - while loop
  - do-while loop
  - nested for loop
  - for-each/advance loop

Nested loop

A loop inside a loop

```
for() {
    for() { }
}
```

* initialization — initial value
* condition — satisfying cond.
* updation — inc/dec.

for-loop

↳ no iteration
↳ repeat statement in a particular range

```
for ( initialization ; condition ; updation ) {
        //instructions }
```

while loop

↳ when no. of iterations are unknown
↳ range is unknown

```
While ( condition ) {
        //instructions
        <updation>
    }
```

ma → ctrl + space (main function)
Sysout → ctrl + space (println statement)

do-while loop

↳ execute a set of instructions atleast once

```
do {
    //instructions
} while (condition);
```

Bansal

DATE
PAGE

Assignment - (I)

Marks validation

marks	category
81 - 100	distinct
35 - 80	First class
0 - 34	oops :(fail

Assignment - (II)

Salary	Employee
50,000 - 1,00,000	Manager
25,000 - < 50,000	Tester
10,000 - < 25,000	HR
0 - 10,000	Helper

Assignment - (III)

```
Char ch = '$';
Switch() {
    character
    number
    special character }
```

Assignment - (IV)

```
Switch() {
    add
    sub
    mul
    div }
```

Isha Bansal

DATE
PAGE

Date - 02-08-2022
Tuesday

Instructor - Manju

Introduction to Methods

- Block of instructions to perform a specific task

<access specifier> <modifier> <return type> <name of methods> (---) { }
↳ formal arguments

Example: public static void hello ( ) { }

What are access specifiers?

- Keywords
- should be in lower case
- responsible for accessibility of a method

(i) private - only within class, not available outside class
(ii) default - within class, within package, not ava. outside pkg
(iii) public - within class, within package, outside package
(iv) protected - within the class, within package, outside with help of child class

	within class	within package	outside package	with child class
private	✓	×	×	×
default	✓	✓	×	×
public	✓	✓	✓	✓
protected	✓	✓	×	✓

What are modifiers?

- Keyword, lowercase
- modify characteristics of the members.

(i) static
(ii) abstract
(iii) final
(iv) synchronized
(v) volatile
(vi) transisent

non-Static
- if modifier is not mentioned automatically non-static

Ba Bansal

DATE
PAGE

static modifier - directly related to the class
abstract modifier - no body, ends with semi-colon (;)
final modifier - not able to inherit
synchronised modifier - will be used in multithreading

main method is always static method

What is return type?
- Type of data which a method returns
void → return nothing

primitive datatype ⇒ if method returns primitive value
non-primitive datatype ⇒ if method returns non-prim. value

Naming convention for method name
→ uppercase
isha Bansal
lowercase

Example:
```
public static void WaterBottle () {
        // instructions
}
```

You should not create any method inside another method.

Types of methods
(i). No argument method
(ii) Parameterized method

x— Code (i) and (ii) —x

Isha Bansal

method declaration – formal arguments
method call – actual arguments

Introduction to Method Overloading

* Class has multiple methods with same name but difference in arguments

Rules of method overloading

One of the rules must satisfy for method overloading

> Methods have different no. of arguments
> Methods have arguments of different data type
> Methods have different sequence of arguments

Static and static members

– having static modifier

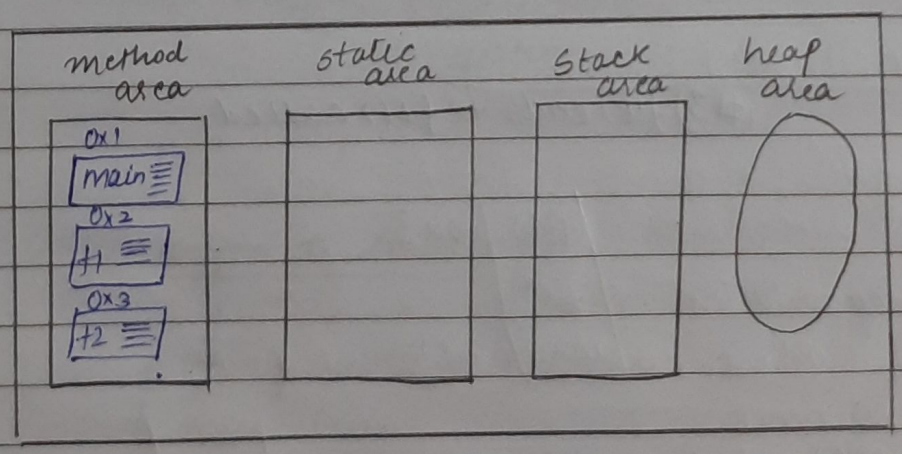
static variable is global in nature

prefixed with 'static' keyword

static members → variable
→ method
→ initializers
↳ static {
//instructions
}

(all initializers are executed before main method)

Understanding execution of Java program.



CLASS LOADING PROCESS

Bansal

Method area

- stores / loads all the methods along with their members and instructions set
- also contains address of the method

Static area

two comp.
* Class static pool
  - ↳ method signature
  - ↳ address of method
* Static variables

M. Sig.	Addr.
main	0x1
f1	0x2
f2	0x3

Stack area

- helps in execution of program

let's consider the order of execution of methods is:

main() ⟶ f1() ⟶ f2()

#1

main() executes

main	0x1

Stack area

» stack area refers to class static pool to get the method signature with help of address and then loads the whole method from method area.

#2 f1() called

f1()	0x2
main	0x1

#3 f1() ends + f2() called

f2()	0x3
~~f1()~~	
main	0x1

#4 f2() ends + main() ends

~~f2~~	0x3
~~main~~	0x1

Isha Bansal

Heap area

- objects are stored inside heap area

DATE
PAGE

Introduction to class

- blueprint of object
- non primitive
- we can have multiple objects for a single class.

instance of a class

all objects created are stored inside the Heap area

attributes
properties of the object
variables

behaviour
functionalities of the object
methods

Steps to create an object

<class Name> <object Name> = new <class Name> ();

reference variable

class Book { }

Book b = new Book();
b.<method>();
b.<variable>;

Introduction to non-Static members

- local to class
- not prefixed by Static modifier

Variables, Method, Initializer, Constructor

can be accessed after creating object of the class

Syntax:
{
// instruction
}

Executed the moment obj of class is created.

Introduction to constructors

- Syntax name of constructor = name of class

[No argument Constructor
parameterized constructor

Isha Bansal

Constructor overloading

- Same rules of method overloading apply.

Constructor chaining

- calling one constructor inside another constructor

Rules

(i) Always takes place inside constructors
(ii) chaining statement is the first statement
(iii) Recursive call to a constructor is not allowed
(iv) atleast one constructor must have 'this' call statement.

```
class C { int a;
    C(int a) {
        this.a = a }
    C(int a, int b) {
        this(a);
        this.b = b; }
}
```

Constructor chaining

Object Oriented Programming

Principles of OOP: Encapsulation
Abstraction (achieved with interface)
Inheritance
Polymorphism

Isha Bansal

DATE
PAGE

Introduction to Encapsulation

An object has two things : attributes & behaviour.
The binding of an object's attributes to its behaviour is known as encapsulation.

Advantages ⇒ DATA HIDING
- restrict direct access
- private Modifier is used
- are directly accessible only within class.

Getter & Setter Method

Help in fetching & updating values of the private attributes of a class.

Getter Method — Get/Fetch values
Setter Method — Set/Update values

These two functions can achieve Verification & Validation.

> The first rule of encapsulation is ⇒ No Constructor
> Getter & Setter functions are created for all private attributes

Getter Method

```
<access specifier> <returntype> get<variable Name>() {
        return variable name; }
```

Setter Method

```
<access specifier> void set<variableName>(<datatype> <val>)
{
        this.<variable Name> = val;
}
```

Bansal

DATE
PAGE

Date → 03-08-2022
Wednesday

Intructor – Manju

Introduction to Relationship

- Association between two objects
  - has-a Relationship
  - Is-a Relationship

Has-a Relationship

↳ one object is dependent on another object
↳ Based on the dependency level
- Aggregation — weak dependency → object can exist without each other
- Composition — strong dependency → object can't exist without each other

Aggregation Examples

↳ Sam has a laptop
Obj ① ← weak → Obj ②

↳ Sunny has a book
Obj ① ← weak → Obj ②

Composition Examples

↳ John has a heart
strong

↳ Human has Oxygen
strong

Isha Bansal

Is-a Relationship

↳ Association b/w objects is similar to parent child relationship

Parent ↑ inherits properties from Child

Parent can't access child's properties.

Is-a relation

- Cat is an animal
- Harry is a human

Is a relation

Note Getter-Setter method are compulsory for encapsulation but for relationship 'constructor' are used to 'set' values and 'Getter' are used to get/fetch values.

Introduction to Inheritance

Base / Super / parent class	Child / Sub / Derived class
- class from which the child inherits properties	- class which inherits properties from the parent class.

X `int i;` declared variables = 1, inherited variables = 0

↑

Y `int j;` declared variables = 1, inherited variables = 1

All attributes of parents → Child implicitly.

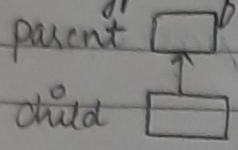
> Inheritance makes use of `extends` keyword

<child-class> extends <parent_class>

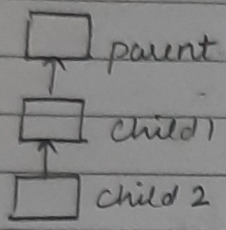
Isha Bansal

DATE
PAGE

Types of Inheritance

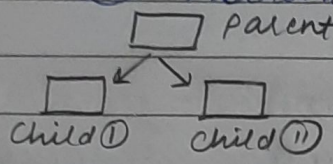


(i) Single Inheritance
- One parent + one child



(ii) Multi-level Inheritance
- chain of parent-child pairs

(iii) Hierarchical Inheritance



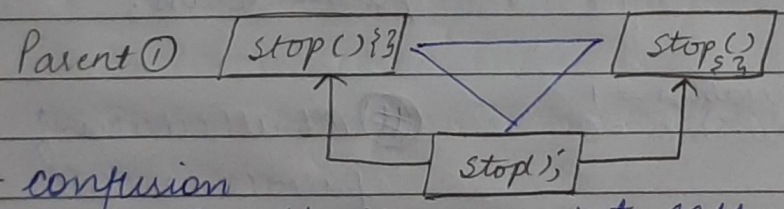
- One parent two children

(iv) Multiple Inheritance] can't implement
(v) Hybrid Inheritance] using classes
↳ Need Interfaces

Problem with Multiple Inheritance

Two parent + One child

- problem occurs when methods of same signature occur in both parents
- ambiguity problem



- diamond problem

- confusion b/w which stop() method to call.

Hybrid Inheritance

- combination of types of inheritance
  - → hierarchical inheritance
  - → multiple inheritance

Isha Bansal

DATE
PAGE

Continued Type Casting

change one data type to another data type

- primitive Type casting
  - widening
  - narrowing
- non primitive Type casting
  - upcasting
  - downcasting

Introduction to upcasting & downcasting

- They are type of object type casting

Upcasting (Child to Parent)
Downcasting (Parent to Child)

UPCASTING	DOWNCASTING
Child to parent	Parent to child
implicitly or explicitly	only implicitly

PARENT
upcasting ↑ ↓ downcasting
CHILD

Introduction to Upcasting

Also known as Generalization

↳ child object is type casted to a parent class object
↳ helps access the variables & methods of parent class to child class.
↳ Access only specific variables or methods

<parent> p = new child();
child object
assigned to parent object

↳ # we can access only parent class attributes as it has been upcasted to parent level.

attributes of child class is not accessible ←

Reason – Upcasting is used when there is a need of code where only Parent class is required.

Isha Bansal

Introduction to Downcasting

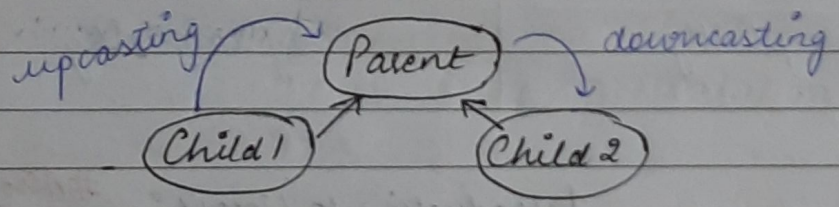
- upcasting is necessary
- converting parent class to child class
- we can't convert parent → child implicitly
  ↳ throws class cast Exception
- explicit downcasting syntax:

<child classname> (c) = (<child classname>) (p);

child reference object variable — parent reference object variable

Reason – When we need to develop a code that accesses behaviour of child class

Class cast Exception

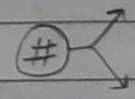


if child 1 is upcasted to parent and only child 1 can be downcasted

Downcasting parent to child2 will give class cast exception.

Introduction to Polymorphism

Poly ⊕ morphism
↓ Many ↓ forms



Compile Time Polymorphism
Binding achieved (C.T.)
+
Same behaviour (R.T.)

Run Time Polymorphism
Binding achieved (Compile Time)
+
Different behaviour (Run Time)

Isha Bansal

Method Binding

- method declaration
  (P) method definition

PSV m1() {} $\xleftarrow{\text{binding}}$ m1()
PSV m2() {} $\xleftarrow{\text{binding}}$ m2()
PSV m3() {} $\xleftarrow{\text{binding}}$ m3()

If m1 function is binded with m1 declaration at compile time and on calling m1, m1 gets executed
↳ compile time polymorphism

If m1 function is binded with m1 declaration at compile time and on calling m1, m3 gets executed
↳ run time polymorphism

Compile time polymorphism

- ✓ Method Overloading
- ✓ Constructor Overloading
- Operator Overloading (X) not supported in Java

Runtime time polymorphism

- method Overriding / dynamic binding
  - ↓ is-a relationship (two required conditions)
  - ↘ upcasting

* If parent & child have a method with same name → method of child class overrides the parent method and it is executed if the function is called.

Ishita Bansal

DATE
PAGE

Introduction to Object class

- parent class of all classes in java
- topmost class in java
- it is present inside $\boxed{\text{java.lang}}$ package
- has non-static methods
- all classes are implicitly extend to the object class.

Methods in Object class

3 main functions
- public String toString()
- public boolean equals(Object O)
- public int hashcode()

$\boxed{\text{toString Method}}$

- returns string representation of this class

$\boxed{\text{equals Method}}$

- compares two variables
- returns boolean value

Requires 3 steps

①

Isha Bansal

DATE
PAGE

Date - 04-Aug-2022
Thursday

Instructor - Manju

Introduction to Abstraction

- Hiding the implementation and provide the functionalities.

How to achieve abstraction in Java?
abstract classes and interfaces

Abstract Classes

↳ can be achieved with the help of abstract methods

Introduction to abstract keyword modifier

↳ abstract public void <method name> ();~~{}~~ → abstract method
↳ abstract method declaration has no set of instructions (ends with a semicolon)

```
abstract class Album{
    abstract public void aname();
    abstract public void asinger();
}
```

```
class song extends Album{
    public void aname(){
        SOP("Alb. name");
    }
    public void asinger(){
        SOP("Singer Name")
    }
}
```

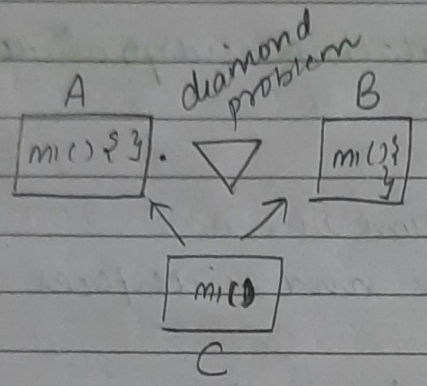
Concrete class

Introduction to Interface

- helps to achieve 100% abstraction
- 'interface' keyword modifier is used.
- only abstract methods are present inside interface
- class is extended from an Interface using the 'implements' keyword.

Isha Bansal

Concept of multiple Inheritance
- does not support with the help of classes.
- can be implemented using interfaces.
  ↳ because no implementation and no constructors



A and B can be:
interface & class
and interface & interface

#1 Parents are interface and class

** implicitly both are taken by the child class.

```
interface I1 {                    Class C1 {
    public void m1();                 public void m1() {
    }                                     SOP("Inside parent class"); }
```

** first class then interface

```
class Child extends C1 implements I1 {
    [ public void m1() {
          SOP("Inside child class"); }
```

overriden!

```
class Driver {
    Child c = new Child();
    c.m1();

    # upcasting
    I1 ic = new Child();
    ic.m1();
```

Isha Bansal

DATE
PAGE

#2 Both parents are Interfaces

```
Interface I1 {
    void m1();
}

Interface I2 {
    void m1();
}

class Child
    implements I1, I2 {
    void m1() {
        SOP("Hi");
    }}

class Driver {
    I1 ico1 = new Child();
    I2 ico2 = new Child();
    ico1.m1();
    ico2.m1();
```

Introduction to Arrays

↳ continuous block of memory
↳ store multiple values

#creation method (I)

```
<datatype>[] a = new <datatype>[size];
int[] a = new int[10];
```

a ⇒ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

↳ implicity a continuous block of memory is created. with default value of datatype as initial value.

size of array = 10
index value = 0 to 9 (9)

a[<index>] = <value>
↳ assign value to the index value

#creation method (II)

```
<datatype>[] a = {e1, e2, e3 ....}
int[] a = {23, 64, 18, 29}
```

a ⇒ | 23 | 64 | 18 | 29 |

Isha Bansal

a.length
↳ returns length of the array.

Introduction to Collection
↳ To store multiple elements together

Problems with array
↳ size of array is fixed
↳ homogenous elements
↳ adding/removing elements is complex.

↳ Collections help overcome the disadvantages of array.

Introduction to Data Structures
↳ used to store and access multiple data together

* Arraylist
* Sets
* Trees
* Graph
* Queue
and many more..

List Data Structure
- store multiple elements
Characteristics of list
↳ maintains an insertion order
↳ allows duplicate values
↳ doesn't have a removal order

Set Data Structure
↳ no duplicate values allowed

Isha Bansal

Queue data structure

↳ first in first out (FIFO)
↳ last in last out (LILO)

Stack data structure

↳ first in last out (FILO)
↳ last in first out (LIFO)

	duplicate values	Insertion order
list	✓	×
set	×	randomly
queue	✓	FIFO
stack	✓	LIFO

Introduction to collection framework

↳ set of classes and interfaces
↳ CRUD operation

C - create
R - read
U - update
D - delete

Collection itself is an interface
↳ defined in java.util package.
↳ non primitive

Collection Hierarchy

Iterable Interface
↑
Collection Interface
↗ List Interface
↑ Set Interface
↖ Queue Interface

List Interface
#class ↳ ArrayList
↳ LinkedList

Set Interface
#class ↳ HashSet
↳ LinkedSet
↳ TreeSet

Queue Interface
↳ priority queue
#class

Ishu Bansal

DATE
PAGE

Common methods to all Collections

* add (object)
* addAll (collection)
* remove (object)
* removeAll (collection)
* clear();
* contains (object)
* containsAll (collection)
* iterator()
* foreachloop()
* size()
* isEmpty()
* hashCode()
* equals()

List Interface
↳ contains two classes that provide implementation → Arraylist, linkedlist

Arraylist

- concrete implementing class to list interface
- present in java.util package
- store multiple elements
- allows duplicate elements

Create object of Arraylist

```
Arraylist a = new Arraylist();
```

↳ size is not mentioned so by default a size of 10 elements is created.

#1 Adding elements

```
a.add (<value>)
```

↳ If we try to add a 11th element

Arraylist are resizable/growable array.

normal array → IndexOutOfBound Exception

Arraylist → array gets extended to the following amount

$$\left(\frac{cap_{current} \times 3}{2}\right) + 1$$

Isha Bansal

DATE
PAGE

Methods of Arraylist

(i) a.add (<value>)

a.add (<index>, <value>)
↳ if <index> is more than current capacity
↓
IndexOutOfBound Exception

(ii) a.size()
↳ returns the length of the array.

(iii) a.get (<index>)
↳ return the element at the index value
↳ if <index> is not present
(IndexOutOfBound Exception)

(iv) a.contains (<value>)
↳ returns boolean value
↳ if <value> present in array ⇒ True
otherwise ⇒ false

(v) a.remove (<index>)
↳ remove the element at a particular index.

(vi) a.isEmpty() - boolean value
- if array is empty ⇒ True
otherwise ⇒ false

(vii) a.clear() - clear all elements of the array.

a.indexOf (<value>) [index value of first occurrence
if value not present ⇒ -1

(viii) ~~a.addAll(a1)~~

(ix) ~~a = [1, 2, null, "first"]~~
~~a1 = [3, false, "second"]~~

(x) a.lastIndexOf (<value>) - index value of last occurrence

Isha Bansal

a1 = [1, 2, 3, 7, 8]
a2 = ["hi", "hello", "hola"]

a1.addAll(a2) => [1, 2, 3, 7, 8, "hi", "hello", "hola"]
a1.add(a2) => [1, 2, 3, 7, 8, ["hi", "hello", "hola"]]

INTRODUCTIONS TO ITERATORS

↳ present inside java.util package
↳ is used to access the elements
↳ Iterator interface

Iterator interface
↑
List Interface

* hasNext() method
* next() method
* remove() method

hasNext() Method

hasNext() → false

["Java", 1, 2, 3, 6.5f, 3.5]

↑ hasNext() → true

If there is another element present after the current element → return true
Otherwise → false.

Next() Method

This will return the ~~next~~ current element to the current pointing element. and makes the cursor move forward.

remove() Method

Remove the current element & move cursor forward.

Isha Bansal

DATE
PAGE

Disadvantages of Iterator

- we can iterate only once
- reverse order not possible
- add / remove elements not possible

hasPrevious()
previous().

ListIterator

Introduction to for-each loop

```
for (data type var name : collection object)
{
    SOP(varname);
}
```

If no fixed datatype
↳ Object is used.

Introduction to linked list

```
LinkedList LL = new LinkedList();
```

element	addr of next node

Node

LL.add(<value>)

Introduction to Set Interface

↳ no duplicate values
↳ random order

* HashSet
* LinkedHashSet
* TreeSet

Implementation of Set Interface

HashSet

- concrete implementing class

Characteristics

↳ unordered
↳ no duplicate
↳ no index
↳ heterogenous elements

Isha Bansal

DATE
PAGE

```
HashSet hs = new HashSet();
```

Key	value
001	10
002	20
003	30

```
hs.add(10);
hs.add(20);
hs.add(30);
```

Add more elements beyond available space

75% of cap is extended.

```
h.add(x1)
h.add(x2)
h.add(x1)
h.add(x3)
h.add(x4)
```

key	value
001	x1
002	x2
003	x3
004	x4

Introduction to Linked Hash Set

- maintains an insertion order

Introduction to Treeset

- concrete implementing class of set interface
- sorted order

Isha Bansal

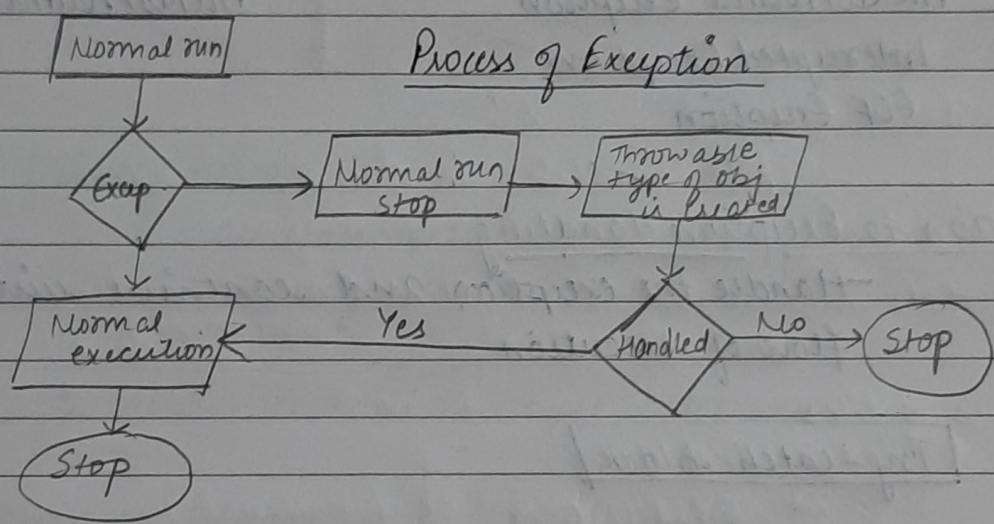
Date - 05-Aug-2022
Friday

Instructor - Manju

Introduction to Exceptions

- unwanted event that stops the execution of the program.
- unlike error, we can handle exceptions

Process of Exception



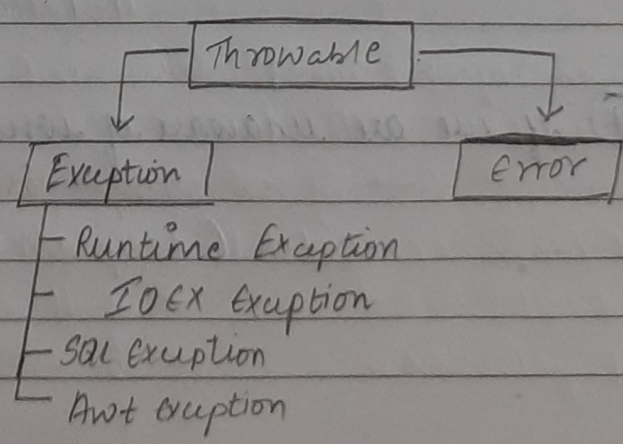
* Exception is generated by one statement.

```
int a=10;
int b=0;
SOP(a/b);
```
→ Arithmetic Exception

Types of Exception
(I) checked exception
(II) unchecked exception

handle immediately
Checked exception — compiler is aware of the exception
unchecked exception — compiler is not aware of the exception

The 'throwable' parent type is present in java.io pkg



Shri Bansal

DATE
PAGE

Run-time Exception

- Arithmetic Exception
- IndexOutofBound Exception
- NoSuchElement Exception
- and many more.

] unchecked exception

IOEX exception

- FileNotFound Exception
- Interupted Exception
- EOF Exception

Awt Exception

- FileNotFound Exception

Introduction to Exception Handling

- Handle the exception and continue with normal flow of execution

Try-catch block

(I) try block

```
try
{
    //statement which is throwing exception
}
```

(II) Catch block

It will catch the exceptions

```
catch (<name of exception> e)
{
    //statements
}
```

(#) If we are unaware of what exception might occur.

parent class of all exceptions.

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

Isha Bansal

DATE
PAGE

printStackTrace — prints details/information about the exception w/o interrupting the normal flow of execution.

getMessage — Provides only small information about the exception instead of complete details.

```
catch (Exception e)
    → e.printStackTrace();
    → SOP(e.getMessage());
```

Finally block

- all statements under 'finally' block are always executed

It is possible to have multiple catch blocks after try block

Thread.sleep(2000); → checked exception
↳ Interrupted Exception.

File Handling in Java

```
import java.io.FileReader;

FileReader fr = new FileReader("text.txt");
```

↳ error because file doesn't exist in system.

Throws Keyword

↳ indication for caller about exception
↳ generally used with Checked exception.

helps to throw exception from a method to main method.

Isha Bansal

DATE
PAGE

finally	finalize	final
		(variable)
set of instructions always executed	it is a method in java. when object is removed from garbage collection	Re-initialization is not possible
		not able to override (method)
Method for exception handling	Method for garbage collection	not able to inherit (class)

File Handling in Java

File – data stored on the system (music, audio, video document)

Types of files
- (i) Text files
- (ii) Binary files

TEXT FILE – files which we can read & write easily. (.java, .txt)

BINARY FILE – files which we can't read & write easily (.class, .jpg)

File f = new File (<address of file>)

create object : File f = new File ("Hello.txt");
f.exist(); → True/False
↳ If not present : FileNotFoundException
f.canWrite(); & f.canRead();
f.getName(); # f.getAbsolutePath();
f.getParent(); # f.isFile(); # f.createNewFile();
f.isFolder();
100% throw Exception

Isha Bansal

Auto - closable Interface
↑
closable Interface
↗ ↖
Input Stream Output Stream

Input Stream
Writer class
- File writer
- Buffer reader

Output Stream
Reader class
- File reader
- Buffer writer

file writer doesn't support newline
file reader will return in type of integer

```
fileWriter fw = new FileWriter(f);
FileReader fr = new FileReader(f);

fw.write(<msg>);
fr.read();  → int → ascii value
(char) fr.read();  → character
```

```
int V = fr.read();
while (V != -1) {
        SOP((char) V);
        V = fr.read(); }
```

Overcoming disadvantages of FileReader and FileWriter

Buffered Writer
↳ can add newline using newline method.

Buffered Reader
↳ read line by line instead of character by character.
↳ return type of readline method is String

Ishita Bansal

DATE
PAGE

Java 8 Features

↳ stable

↳ functional Interface

Functional Interface

- if interface has only one abstract method

↳ default method

Default Method

- have concrete method inside interface with the help of 'default' keyword.

↳ method reference

Method Reference

- Interface must be functional Interface.

```
interface I1 {
    void fun1();
}

class Driver {
    public static void m1() {
        SOP("Hello");
    }
    PSVM() {
        I1 i = Driver::m1;
        i.fun1();
    }
```

connected m1 with fun1.

Isha Bansal

DATE
PAGE

LAMBDA EXPRESSION

- functional interface is mandatory
- give implementation w/o object.

```
I, iL = [() ->] → lambda
{
    SOPL("hello");
};
```

Introduction to Multi Threading

→ Each task has a number of sub processes.

Process - Sub tasks in achieving a particular tasks

For each task happening simultaneously is assigned a thread

- → smallest unit of execution
- → light weight process
- → Independent of other threads

main / default thread
↳ whenever we execute a java program this thread is executed

Priority level of thread (1-10 (range))
↳ 5 (default)

Why do we need threads?

Sister is asking for 12k
Mother is asking for 12k
Bank balance ⇒ 15k

Your sister is swipping your card and at the same time you paytm your mother

major disadvantage
both successful

Isha Bansal

In the concept of multi threading ⇒ while your card is being swipped for money, your paytm transaction needs to wait and vice versa.

Multi Threading

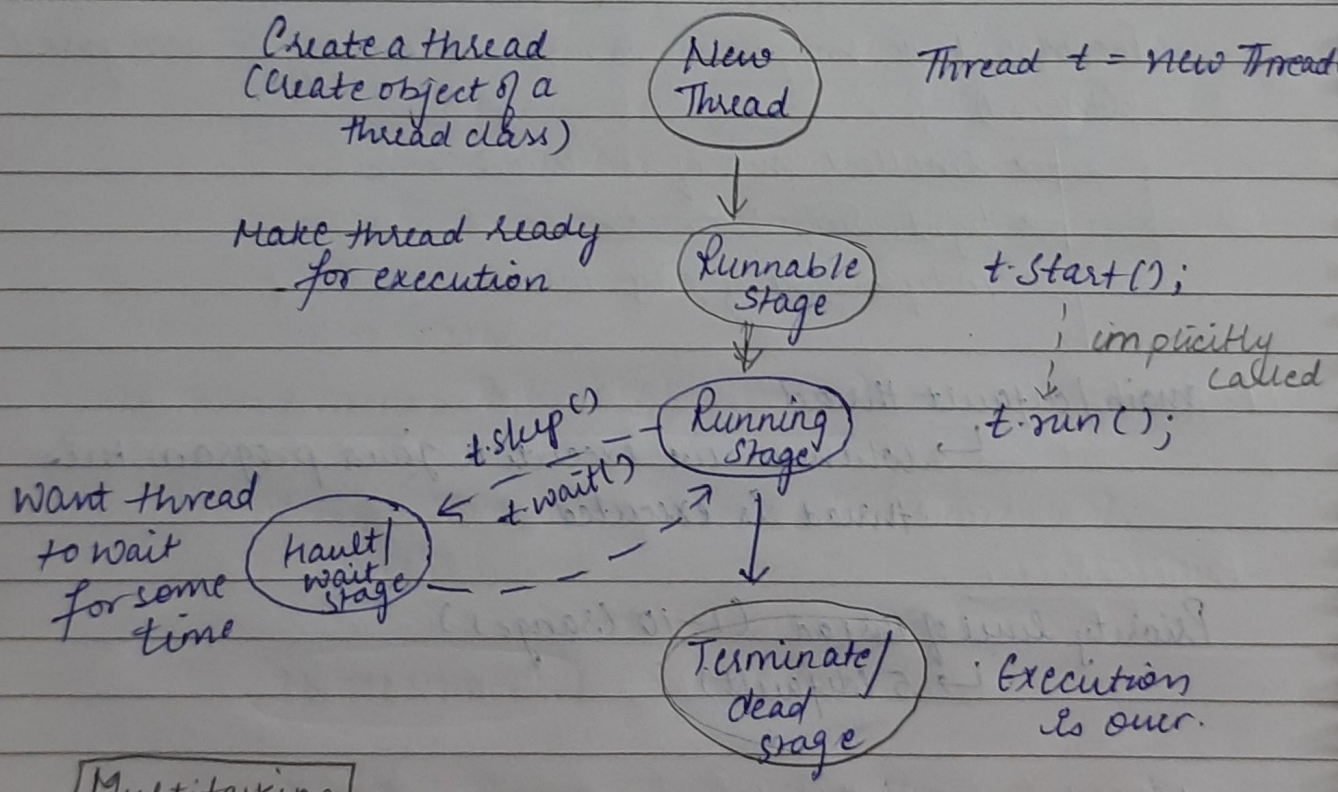
Process of executing multiple threads at a time

How to create a thread

- Thread class
- Runnable Interface

Building a house alone ⇒ 5 years
——— with 100 ppl ⇒ 1 year
——— 500 ppl ⇒ 3 months

Lifecycle of Thread



Multitasking

- executing multiple tasks at a time

- Process based M.T.
- Thread based M.T.

Isha Bansal

Process based Multitasking

↳ performing several tasks simultaneously

↳ each task is independent

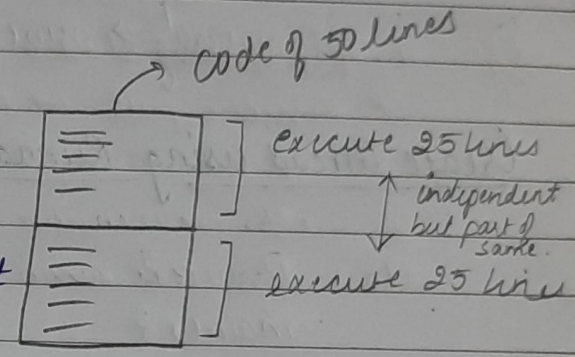
→ independent tasks

[programming code ⊕ listening to music

Thread based Multitasking

↳ executing several task simultaneously

↳ each task is independent but are sub tasks of a bigger task



Some methods in Multi threading

start() — push thread to runnable stage

run() — run the thread

sleep(<time>) — wait the thread for a certain time
time is given in millisec (ms)

getId() — get ID of a particular thread

getName() — get Name of a particular thread.

setName() ✓

getPriority() ✓

setPriority() ✓

isAlive() — is a thread alive/dead?

Constructors in thread class.

> Thread()

> Thread(String name)

> Thread(Runable r)

> Thread(Runnable r, String name)

Isha Bansal

DATE
PAGE

Create Thread using Thread class

```
class SingleThread extends Thread {
    start()
automatically → run()
```

] already present in thread class → we are overriding both methods

Create Thread using Runnable Interface

```
class Driver {
    main() {
        // create object of class
            ThreadClass tc = new ThreadClass();
        // create object of Thread
            Thread t1 = new Thread (tc);
        // start thread
            t1.start();
    }
}
```

Create Thread using Thread(String name)

```
Thread t1 = new Thread ("FirstThread");
t1.start();

t1.getName();  → FirstThread.
```

Create Thread using Thread(Runnable r, String name)

```
class <class> implements Runnable {
main(){Runnable tr = new <class>
    Thread t = new Thread (tr, "Name");
    t.start(); }
```

Isha Bansal