

1.Introduction to Java

1.1 Computer Programming

What do you mean by the term program and programming language?

- A Program is a set of instructions that perform a particular task.
- To give instructions to machines we must know machine language.
- But machine language is very difficult to understand.
- Hence programming language is used to bridge the gap between humans and machines.

1.2 Definition of Java

Definition:

Java is an object oriented, platform independent programming language that can be used to develop web, mobile and desktop applications.

- Java was developed in the year _____ by _____.
- Latest version of Java is _____.

Features of Java:

1) Simple

2) Object oriented

3) Open source

4) Platform independent

1.3 Installing Java

Steps to install Java on Windows

- Go to oracle website
<https://www.oracle.com/technetwork/java/javase/downloads/index.html> and click on the download button near oracle jdk.
- Double click the downloaded **jdk.exe** file and complete the installation.
- Goto Java installation directory till bin folder and copy the path.

Ex - C:/Program Files/Java/JDK/bin

Paste the above path in environment variables.

Steps to install Java on Linux

- Run the following commands in your terminal
- **sudo add-apt-repository ppa:webupd8team/java**
- **sudo apt-get update**
- **sudo apt-get install oracle-java8-installer** (Depends on version).

Steps to install Java on Mac OS X

- Go to oracle website
<https://www.oracle.com/technetwork/java/javase/downloads/index.html> and click on the download button near oracle jdk.
- Double click the downloaded **jdk.dmg** file and complete the installation.

1.4 First Program

- Open a text editor like **notepad**, **notepad++**, **sublime** etc. Write the following program.

```
class Hello{  
  
    public static void main(String[] args){  
  
        System.out.println("Hello world");  
  
    }  
  
}
```

- Save the above file with **.java** extension (Ex hello.java).
- Open command prompt / terminal and navigate to the directory where the program file exists.
- Compile the above program using the following command.
- _____.
- Once the java file is successfully converted into .class file(bytecode), execute the above program using the following command.
- _____.

1.5 Comments in java

Why should we write comments in code ?

How to write single line comment in Java ?

//

How to write multi-line comment in Java ?

/*

*/

2. Data types, variables and identifiers

2.1 Data types

- A program needs a lot of data to execute and accomplish a desired task.
- Example- A program needs student's marks in different subjects to calculate their percentage or grade.
- Data is stored in computer memory, data type is used to decide the type of data to be stored and amount of memory required to store the data.

There are 8 primitive data types in Java.

4 Data types are used to represent integers/whole numbers (not decimal).

Data type	Size	Range
byte	1 byte	-2^7 to $2^7 - 1$
short	2 byte	-2^{15} to $2^{15} - 1$
int	4 byte	-2^{31} to $2^{31} - 1$
long	8 byte	-2^{63} to $2^{63} - 1$

2 Data types are used to represent decimal numbers.

Data type	Size	Example
float	4 byte	12.5f, 0.5f
double	8 byte	12.5, 0.5

Character data type is used to store single characters a-z, A-Z, 0-9, and all special characters like @, #, \$ etc.

Boolean data type is used to store only **true** and **false** value.

Data type	Size	Example
char	2 byte	'A', 'a', '1', '#'
boolean	JVM dependent	true, false

String is a non-primitive data type. String is used to store a series of characters.

Example - "Batman", "1234", "Hello123", "@#123"

Choose the appropriate data type for the following data -

123, 100, 56	
200, 500, 1000	
20000, 30000	
'A', '2', '%'	
true, false	
"Hello", "ITVedant"	
350.67, 78.9	
34.5f, 56.8f	

2.2 Variables

- Variables are like containers that store the data.
- In Java, creating a variable consist of 2 steps-
 - 1) _____
 - 2) _____
- Data stored in variables can be changed.

Example -

```
int age = 26
```

```
double height = 162.5
```

```
String name = "Batman"
```

```
boolean isEligible = true
```

In the above example **age**, **height**, **name** and **isEligible** are variables

Declare and initialize at least 10 variables. (Make sure all data types are covered).

[illegible]

2.3 Identifiers

- Identifier is a name given to a variable, class or method.
- Identifier rules
 - 1) Should consist of 'a-z', 'A-Z', '0-9' and only two special characters '\$' and '_'.
 - 2) Should not start with a number.
 - 3) Case sensitive.
 - 4) Can have infinite length.
 - 5) Should not be a keyword.

Valid identifiers	Invalid identifiers
firstname	first name
\$firstname	1_num
first_name	num#
firstName	Num&
num_1	2_num
num_2	class
isEligible	int

check if the following identifiers are legal or not (if not why?)

Identifier	Valid (Tick mark)	Not valid(Reason)
num_1		
9name		
double		
\$mathScore		
num&		

3. Type Casting

Changing the data type of a given variable is known as type casting.

3.1 Numeric type casting

Implicit casting

Moving from lower (in size) data type to higher data type is known as implicit casting.

byte -> short -> int -> long

```
byte b = 10;  
  
short s = b;  
  
int i = s;  
  
long l = i;  
  
System.out.println(l);  
// 10
```

Implicit casting is simple as data keeps moving to higher memory size.

Explicit casting

- Moving from higher (in size) data type to lower data type is known as explicit casting.
- Since data moves from higher memory size to lower memory size, there is a chance of data loss and compiler throws an error.
- Hence developer must cast explicitly accepting the data loss.

long -> int -> short -> byte

```
int i = 200;  
byte b = i; //error  
  
byte b = (byte) i;  
  
System.out.println(b);  
  
// not 200, because 200 does not fit in the  
range of byte (data loss)
```

Casting between whole numbers and decimal numbers:

```
int i = 10;  
  
double d = i;  
  
System.out.println(d);  
// 10.0  
  
  
double d = 10.5;  
  
int i = (int) d;  
  
System.out.println(i);  
//10 (data loss)
```

3.1 Numeric and Character Type Casting

Yes, it is possible to cast a number to a character and a character to a number, as all the characters have a **unicode**.

```
char a = 'A'

int i = a;

System.out.println(i)
// 65 (Unicode of 'A')

int i = 97;

char c = (char) i;

System.out.println(c);
// 'a'
```

3.3 Numeric and String Type Casting

- Direct casting between string to number is not possible.
- This can be achieved with the help of **wrapper class**.
- Every primitive data type have its own corresponding wrapper class.

Primitive data type	Wrapper class
byte	Byte
short	Short
int	Integer
long	Long
double	Double
float	Float
char	Character
boolean	Boolean

Example -

```

String num_1 = "123";

int i = Integer.parseInt( num_1 );

System.out.println(i);
// 123

String name = "batman";

int i = Integer.parseInt( name );

// Throws exception as "batman" cannot be converted into a number

```

Complete the exercise -

Given	Instructions	Answer
int a = 10;	Convert to byte	
float f = 12.5f;	Convert to int	
byte b = 100;	Convert to short	
String s = "45";	Convert to int	
String s = "45.7";	Convert to double	
char c = '2';	Convert to int	
char a = 'A';	Convert to int	
short s = 123;	Convert to long	
int a = 200;	Convert to String (bonus)	
double d = 23.5;	Convert to String (bonus)	

4. Operators

4.1 Arithmetic operators

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus (outputs remainder)
++	Increment
--	Decrement

Consider two variables and complete the exercise

```
int a = 4;  
int b = 6;
```

Expression	Output
a + b	
a / 2	
a % 2	
b * 3	
a ++	
b --	

b / a	
a - 3	
b % a	

Pre and post increment / decrement

Pre increment

```
int a = 10;
System.out.println( ++a ); //pre increment
```

Output - 11

Post increment

```
int a = 10;
System.out.println( a++ ); //post increment
```

Output - 10

```
System.out.println( a );
```

Output - 11 // Value increases at the next line

The above concept applies exactly same for decrement.

4.2 Comparison operators

Comparison operators always returns a true or false value

Operator	Description
>	Greater than
<	Lesser than
>=	Greater than or equal to
<=	Lesser than or equal to
==	Equal to
!=	Not equal to

Complete the exercise

Expression	Output (true / false)
4 > 2	
2 > 3	
5 < 3	
4 < 8	
7 == 7	
6 == 8	
4 != 8	
4 != 4	
4 >= 2	
5 <= 4	

4.3 Logical operators

Logical operators are used to combine multiple comparison expressions

Operator	Description
&&	AND
	OR
!	NOT

Complete the exercise

Expression	Output (true / false)
4 > 2 && 5 > 3	
2 > 3 && 1 < 4	
5 < 3 3 < 2	
6 > 3 8 < 9	
!(4 > 2)	

Important - The above operators are also applicable on **char** data types. Every characters have **unicode**, hence the above operations are possible on their corresponding unicodes.

4.4 Assignment and shorthand operators

Operator	Example
=	int a = 10;
+=	a += 10 (a = a + 10)
-=	a -= 10 (a = a - 10)
*=	a *= 10 (a = a * 10)
/=	a /= 10 (a = a / 10)
%=	a %= 10 (a = a%10)

4.5 Bitwise operators

Operator	Description
&	AND
	OR
^	XOR
~	NOT / COMPLIMENT
>>	RIGHT SHIFT
<<	LEFT SHIFT

Bitwise operators operates on binary numbers

Number	Binary
0	00000000
1	00000001
2	00000010
3	00000011
4	00000100
5	00000101
6	00000110
7	00000111
8	00001000
9	00001001
10	00001010

Bitwise AND operation

```
int a = 1;  0001
int b = 2;  0010
a & b

  0 0 0 1
  0 0 1 0
  -----
  0 0 0 0

Output- 0
```

Bitwise OR operation

```
int a = 1;  0001
```

```
int b = 2;  0010
```

```
a | b
```

```
0 0 0 1  
0 0 1 0  
-----  
0 0 1 1
```

Output- 3

Bitwise XOR operation

```
int a = 3;  0011
```

```
int b = 5;  0101
```

```
a ^ b
```

```
0 0 1 1  
0 1 0 1  
-----  
0 1 1 0
```

Output- 6

Bitwise NOT / COMPLEMENT operation

```
int a = 5;  00000101
```

```
~a
```

```
0 0 0 0 0 1 0 1  
-----  
1 1 1 1 1 0 1 0
```

Output- (-6)

Bitwise RIGHT SHIFT operation

```
int a = 5; 0101
```

```
a>>1
```

```
0 1 0 1
```

```
-----
```

```
0 0 1 0
```

Output- 2

```
a>>2
```

```
0 1 0 1
```

```
-----
```

```
0 0 0 1
```

Output- 1

Bitwise LEFT SHIFT operation

```
int a = 1; 0001
```

```
a<<1
```

```
0 0 0 1
```

```
-----
```

```
0 0 1 0
```

Output- 2

```
a<<2
```

```
0 0 0 1
```

```
-----
```

```
0 1 0 0
```

Output- 4

Consider two variables and complete the exercise

```
int a = 2;  
int b = 4;
```

Expression	Output
a & b	
a b	
a ^ b	
~b	
~a	
a>>2	
b>>4	
b<<1	
a<<2	

5. Control Structure

5.1 Conditional statements

5.1.1 If statement

- **if** block is used to execute a code only if a particular condition is satisfied
- Condition must return a **true** or **false** value
- Structure of if block -

```
if ( condition that returns true / false ) {  
  
    //statement to execute if condition is true  
  
}
```

Example - To check whether a person is eligible to vote or not. Condition is, age of the person should be greater than 18.

```
int age = 20  
  
if ( age > 18) {  
  
    System.out.println( " Eligible to vote " );  
  
}
```

5.1.2 If - else statement

- **else** block is used to execute a code if a particular condition is not satisfied
- It is followed by **if** block.
- Structure of if-else block -

```
if ( condition that returns true / false ) {  
  
    //statement to execute if condition is true
```

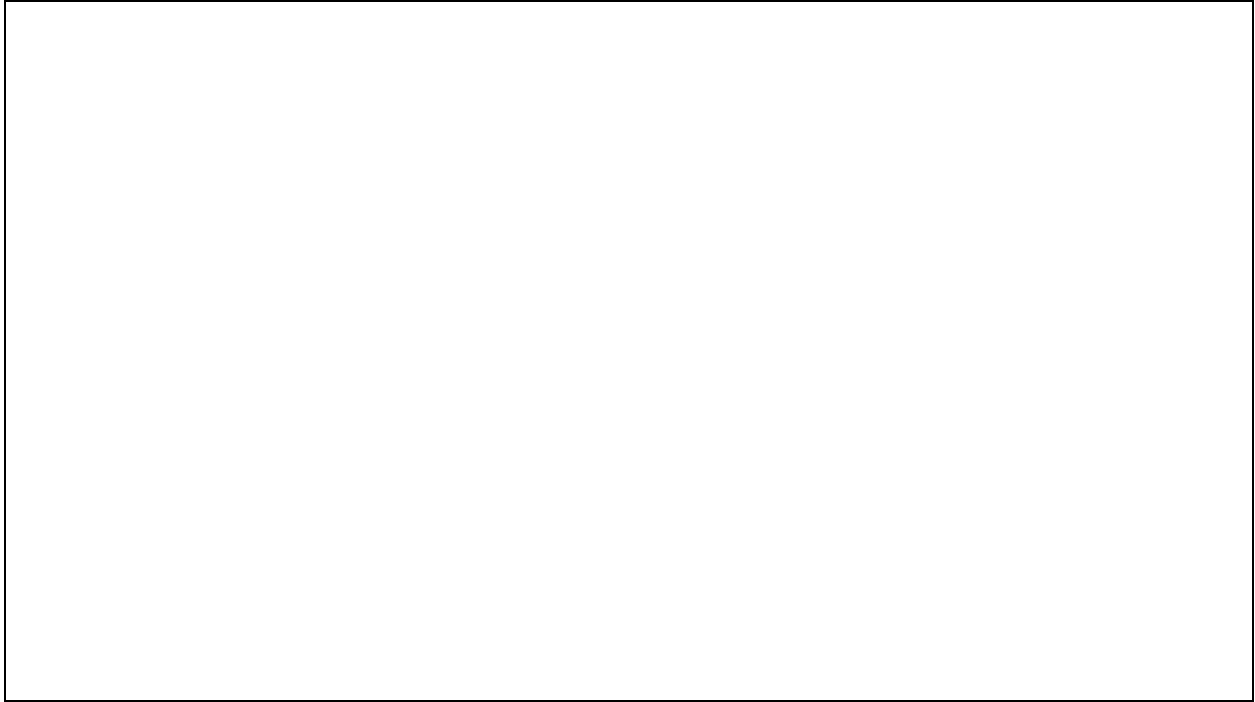
```
} else {  
    //statement to execute if condition is false  
}
```

Example -

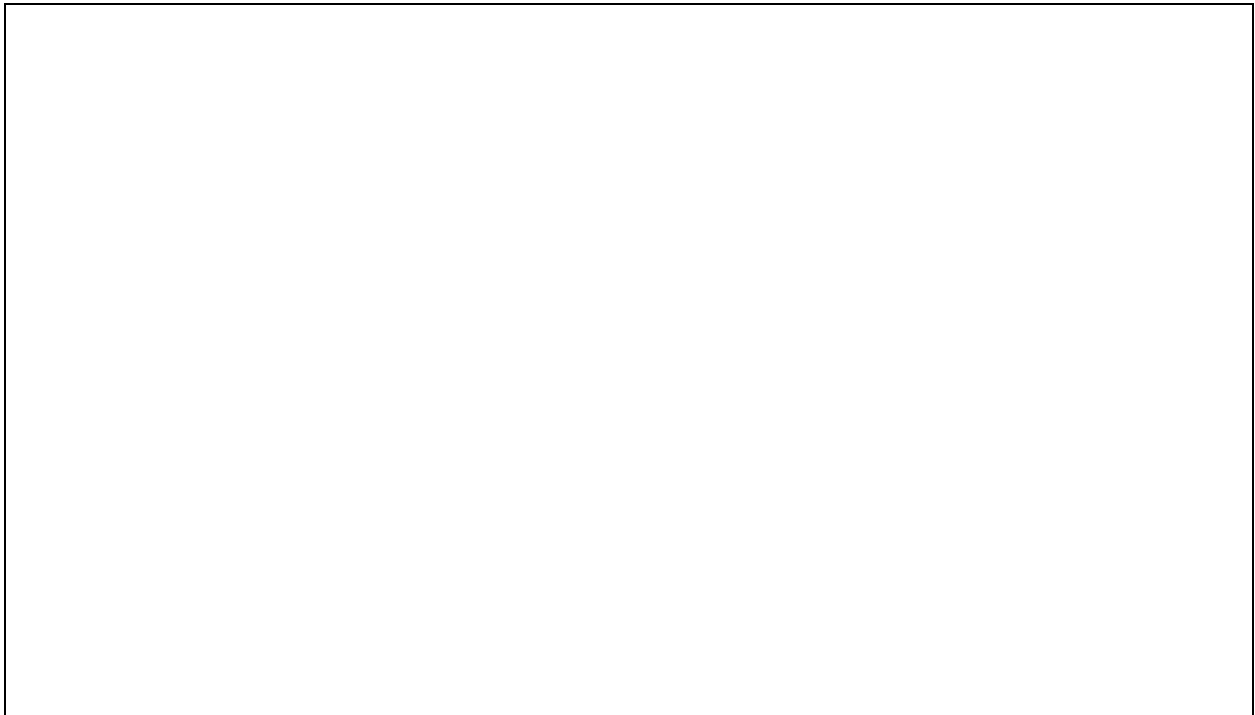
```
int age = 20  
if ( age > 18) {  
    System.out.println( “ Eligible to vote ” );  
} else {  
    System.out.println( “ Not eligible to vote ” );  
}
```

Questions

Write code to check if a number is even or odd



Write code to check if a year is a leap year or not



Write code to weather a character is alphabet or not

5.1.3 If - else - if statement

- **if - else - if** block is used when there are multiple conditions.
- When one of the condition is satisfied then remaining conditions are not checked.
- Structure of if-else-if block

```
if ( condition 1 ) {  
    //statement to execute if condition 1 is true  
  
} else if ( condition 2 ) {  
    //statement to execute if condition 2 is true  
  
} else if ( condition 3 ) {  
    //statement to execute if condition 3 is true
```

```
} else {  
    //statement to execute if none of the conditions are true  
  
}
```

Example - code to check if a number is positive, negative or zero.

```
int num = 10;  
  
if ( num > 0) {  
    System.out.println( "Positive" );  
  
} else if (num < 0) {  
    System.out.println( "Negative" );  
  
} else {  
    System.out.println( "Zero" );  
  
}
```

Questions

Write a code that calculates grade of a student based on his/her percentage

Percentage 80+ -> 'A+' grade

Percentage 70 - 80 -> 'A' grade

Percentage 60 - 70 -> 'B' grade

Percentage < 60 -> 'C grade'

5.1.4 Nested if statement

- Nested **if** block is used when there are conditions followed by other conditions.
- A nested condition is checked only if the outer condition is satisfied.
- Structure of nested if block -

```
if ( condition ) {
```

```
    if ( nested condition ) {  
  
    }  
  
}
```

Example - Check if a person is eligible to join army, condition -> height should be more than 160 cm and percentage should be more than 60%.

```
double height = 162.5;  
double percentage = 85.4;  
if ( height > 160 ){  
  
    If ( percentage > 60 ){  
        System.out.println( "Eligible to join army" );  
    }  
  
}
```

5.1.5 Ternary operator

- Ternary operator is used to convert conditional statement blocks into single line code.
- ? is ternary operator.

condition ? statement if condition satisfies : statement if condition do not satisfies

```
int num = 20;

if ( num % 2 == 0 ) {

    System.out.println( "Even" );

} else {

    System.out.println( "Odd" );

}

//The above code can be converted into a single line with the help of ternary
operator -

System.out.println ( num % 2 == 0 ? "Even" : "Odd" );
```

Questions

Write code to check if a number is divisible by 7 using ternary operator

Write code to check if a person is eligible to vote (age > 18) using ternary operator

5.1.6 Switch statement

- **switch** statement is used to execute a code block based on selected option.
- Structure of switch statement.

```
switch ( choice ) {  
  
    case choice1 : //code if choice 1 is selected  
  
    case choice2 : //code if choice 2 is selected  
  
    case choice3 : //code is choice 3 is selected  
  
    default : //code is wrong choice is selected  
  
}
```

Example

```
int choice = 2;  
  
switch ( choice ) {  
  
    case 1 : System.out.println( "choice 1 is selected" );  
  
}
```

```
    case 2 : System.out.println( "choice 2 is selected" );

    case 3 : System.out.println( "choice 3 is selected" );

    default : System.out.println( "Invalid input" );

}
```

Output of above code is

```
choice 2 is selected
choice 3 is selected
Invalid input
```

Important - *switch block starts executing from the selected option till the end of the block. Hence is important to use **break** keyword after every case. **Break** keyword allows us to jump out of a particular code block. In this case **break** keyword is used to jump out of switch block.*

```
int choice = 2;

switch ( choice ) {

    case 1 : System.out.println( "choice 1 is selected" );
              break;

    case 2 : System.out.println( "choice 2 is selected" );
              break;

    case 3 : System.out.println( "choice 3 is selected" );
              break;
```



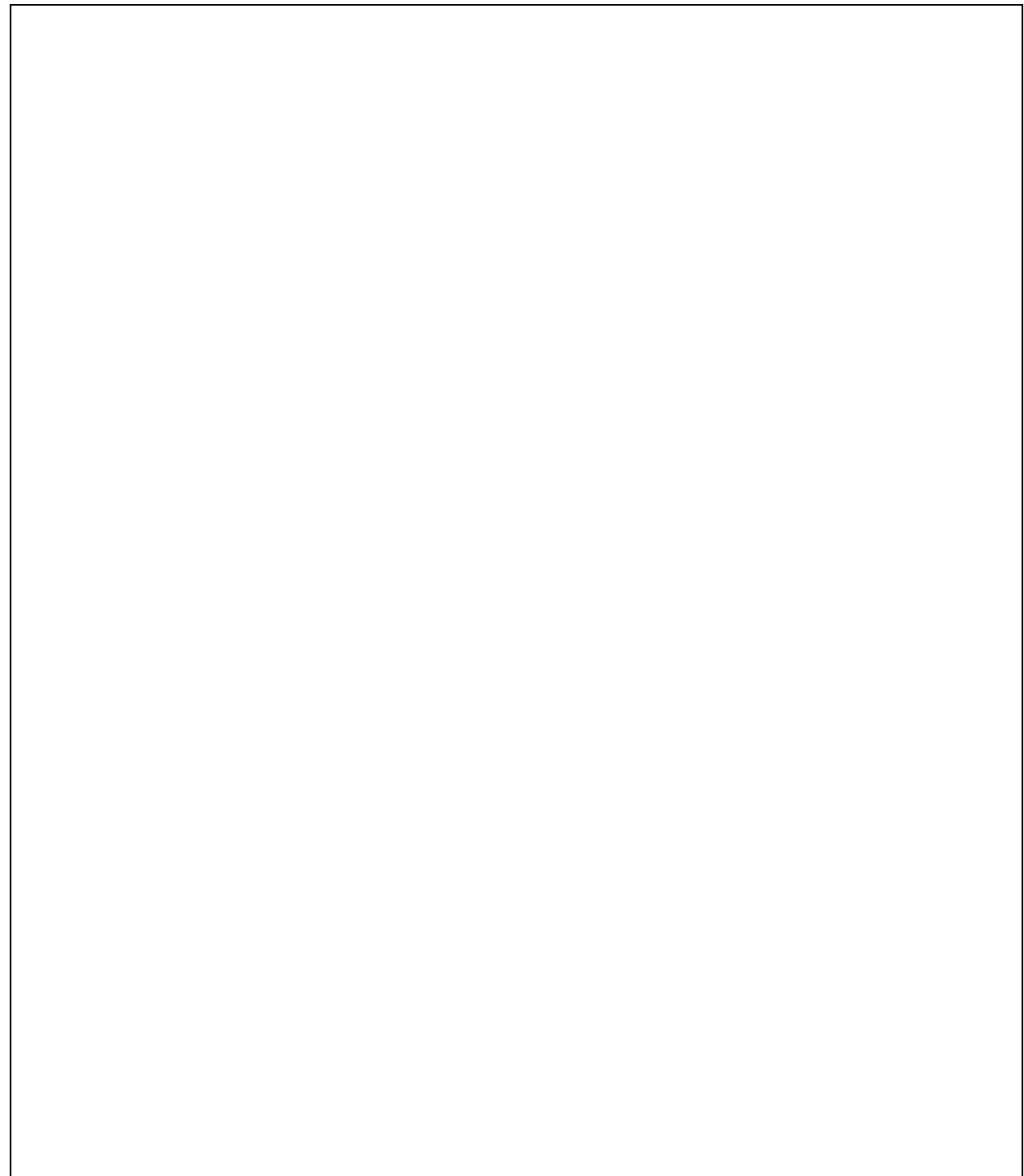
```
        default : System.out.println( "Invalid input" );  
    }
```

//Output of above code is

choice 2 is selected

Questions

Create a calculator using switch statement (Addition / Subtraction / Multiplication / Division) based on user's choice. *Hint - use Scanner class to take user input (ask your trainer for clarity)*



5.2 Loops

- Loops are used to execute a code continuously.
- Loop executes continuously until and unless a condition is satisfied.
- Example searching for a name in a list until and unless the name is found.
- There are 3 types of basic loops in Java - **while**, **do while** & **for**.

5.2.1 While loop

- **while** loop is used when we are not sure of the number of iterations.
- Just like **if** block, **while** block executes when the condition is true.
- Only difference is if block executes only once, but while block executes continuously.
- To stop while loop, condition must become false. This is done with the help of **updating variable** in the loop.
- Structure of while loop block -

```
//Initialization of variable for updation
```

```
while ( condition ) {
```

```
    // code to execute
```

```
    // updating the variable
```

```
}
```

Example - code to print all the digits of a number one by one

```
int num = 121
```

```
while ( num > 0 ) {
```

```
    System.out.println (num%10);
```

```
    num = num / 10;
```

```
}
```

Questions

Write code to find reverse of a number

Write code to check if a number is palindrome or not?

Write code to check if a number is armstrong number?

5.2.2 Do while loop

- Do while loop executes the code and then checks the condition
- Even if the condition is false, do while loop executes at least once.
- Structure of do while loop.

```
//Initialization of variable for updation
```

```
do {
```

```
    // code to execute
```

```
    // updating the variable
```

```
} while ( condition );
```

Example - The code below will execute at least once even if the condition is false

```
do {  
    System.out.println( "Hello world" );  
} while ( false );  
  
// Output  
Hello world (even if the condition false)
```

5.2.3 For loop

- **for** loop is used when we know the number of iterations.
- In for loop- **initialization, condition and updation** is done inside the block.
- Structure of for loop -

```
for ( initialization ; condition ; updation ) {  
  
    // code to execute  
  
}
```

Example

```
for ( int i = 0 ; i < 5 ; i++ ) {  
    System.out.println( "Hello world" );  
  
}
```

```
for ( int i = 0 ; i < 5 ; i ++ ) {  
    System.out.println("Hello world");  
}  
System.out.println( i ); // Error, since scope of the variable "i" is within the body  
                           of the loop
```

Questions

Write code to print all the even numbers between 1 to 50

Write code to print fibonacci series of length 10

Write code to check if a number is prime or not

5.2.4 Nested for loop

- Nested for loop is similar to nested if block, where loop is followed by another loop.
- Structure of nested for loop -

```
for ( initialization ; condition ; updation ) {  
  
    for ( initialization ; condition ; updation ) {  
        // code to execute  
    }  
  
}
```

Example - code to print a right angle triangle


```
*  
* *  
* * *  
* * * *  
* * * * *
```

```
for ( int i = 1 ; i <= 5 ; i ++ ) {  
    for ( int j = 1 ; j <= i ; j++ ) {  
        System.out.print( "*" );  
    }  
    System.out.println();  
}
```

Questions

Write code to print all prime numbers between 1 to 50

Write code to print the following pattern

```
*  
* *  
*  *  
*   *  
* * * * *
```

6. Methods

6.1 Defining and calling method

- A Method is a set of lines of code (or procedure) that perform a particular task.
- The code inside the method executes when a method is called or invoked.
- Structure of a method.

```
access_modifiers  return_type  methodName ( parameters ) {  
  
    // code to execute  
    return_statement  
  
}  
  
//calling the method  
  
methodName( arguments );
```

Important - We will learn about **access modifiers** in depth in upcoming chapters. Till then we will use **public and static** access modifiers.

Example

```
public static void greet ( ) {  
  
    System.out.println( "Hello world" );  
  
}  
  
//calling the method  
  
greet( );
```

6.2 Parameters and arguments

- Consider a method that adds two numbers every time it is called.
- Hence we need to pass two different numbers as inputs every time the method is called.
- This is done by defining **parameters** in the method and passing **arguments** whenever it is called.

Example

```
public static void sum ( int a, int b ) {    // Defining parameters

    System.out.println( a+b );

}

//calling the method

sum ( 2, 3 );    // Passing arguments

sum ( 4, 5 );

sum ( 6, 7 );
```

6.3 Returning a value

- It is not necessary to print a value from the method, sometimes a method can be used just to return a value that can be used by other code.
- This is achieved with the help of **return** keyword.
- If we are returning a value from the method, we must define the **return type** of the method.
- Return type is the data type of the value that is returned from the method.
- If we are not returning anything from the method, return type should be **void**.
- Return statement is end of method body.

Example

```
public static int sum ( int a, int b ) {  
  
    return a + b;  
  
}  
  
//Calling the method and catching the returned value  
  
int value = sum( 4, 6 );
```

Questions

Fill the appropriate parameters, logic and return types for the following problems

Write a method that calculates area of a circle with given radius value

```
public static _____ areaOfCircle ( _____ ) {  
  
    return _____ ;  
}
```


6.4 Method overloading

- Method overloading is a concept where in a class we can create multiple methods with same name, but they must have different number of parameters or different types of parameters or different order of parameters.
- Method overloading never depends on different return type.

Example

```
// Method to return sum of two numbers
```

```
public static int sum ( int a , int b ) {
```

```
    return a + b;
```

```
}
```

```
// Method with same name but returns sum of three numbers (different number of arguments)
```

```
public static int sum ( int a , int b , int c ){
```

```
    return a+b+c;
```

```
}
```

```
// Method with same name but returns sum of two decimals/doubles (different types of parameters)
```

```
public static double sum ( double a , double b ) {
```

```
    return a+b;
```

```
}
```

// Method with same name but different orders of parameters

```
public static double sum ( int a , double b ) {  
    return a + b;  
}  
  
public static double sum ( double b , int a ){  
    return a + b;  
}
```

```
public static int sum ( int a , int b ) {  
    return a + b;  
}  
  
public static void sum ( int a , int b ) {  
    System.out.println( a + b );  
}
```

Important - The code above returns **compile time error**. Method overloading never depends on different **return types**. Hence above code is **not** an example of method overloading.

6.5 Calling method from a different class

Important - As discussed earlier, we will be using **public and static access modifiers**. Calling a **static** method from another class is different from calling a **non-static** method. For now we will learn only about **static** methods. Wait for upcoming chapters for non-static methods.

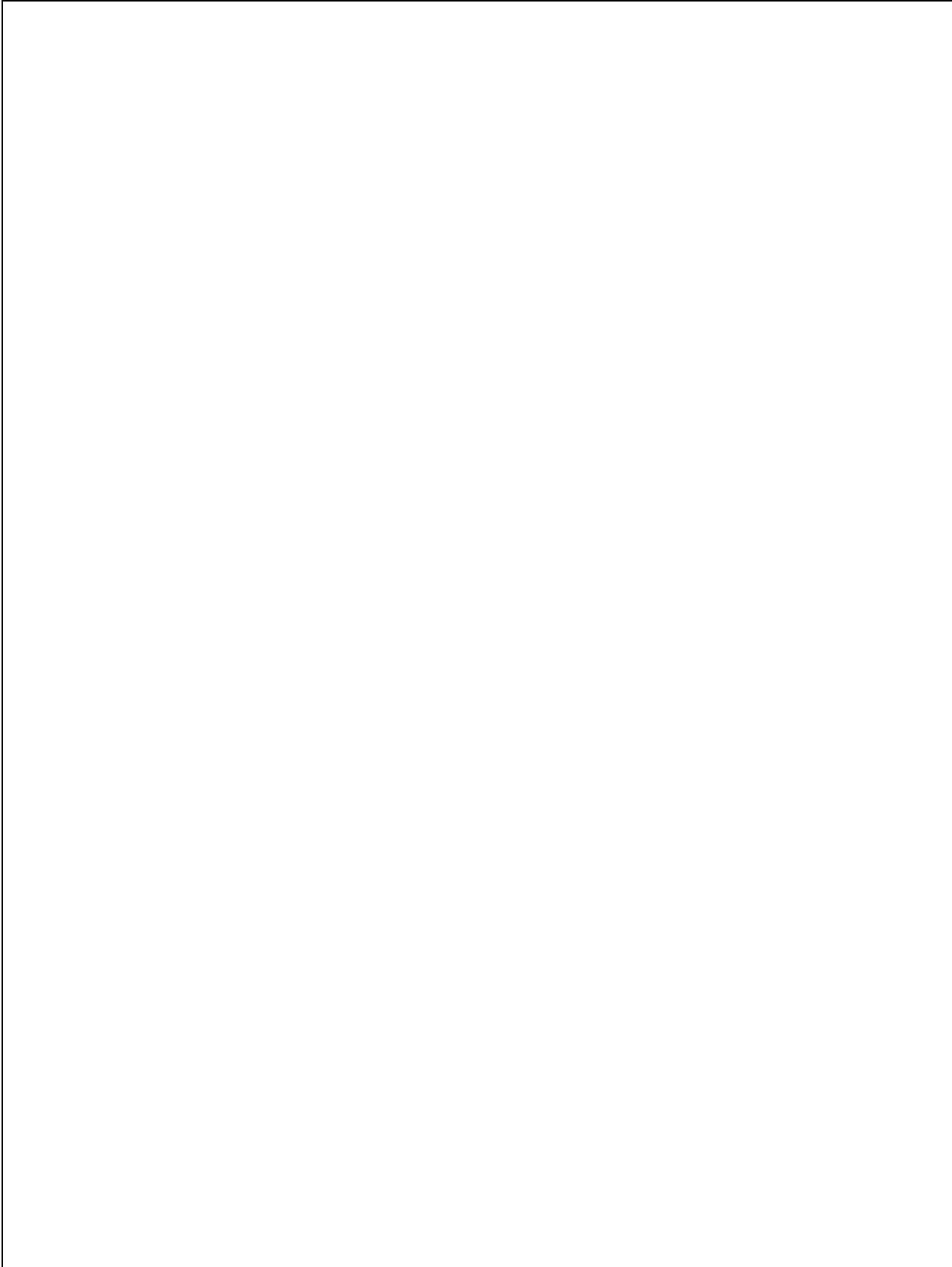
A static method from another class is called with the reference of class name (to which it belongs) using **dot(.)** operator.

Example -

```
class Calculator {  
    public static int sum ( int a , int b ) {  
        return a + b;  
    }  
}  
  
class Test {  
    public static void main ( String[ ] args ) {  
        Calculator.sum( 4,6 );  
    }  
}
```

Questions

Create a class called **Calculator** with methods to **add, subtract, multiply, divide** numbers. Also create at least 2 over **overloaded methods** to operate on different parameters (number, type or order). Call the methods in a different class (Example Test class), that has the **main** method.



7. Arrays

- An array is a homogeneous collection of data in a single variable.
- Elements in an array can be accessed with the help of index.

2	4	6	10	20
0	1	2	3	4

7.1 Creating array

- Arrays can be created using
 - 1) **new** keyword
 - 2) Array initializer
- Once an array is created, it cannot be resized.

Using new keyword

dataType[] arrayName = new dataType[size];

Example

```
int[ ] myArray = new int[ 5 ];
```

```
char[ ] charArray = new char[ 10 ];
```

Using array initializer

dataType[] arrayName = { elements }

Example

```
int[ ] myArray = { 10, 20, 30, 40 };
```

```
char[ ] charArray = { 'a', 'b', 'c' };
```

Arrays generated using **new** keyword consist of default values -

Data type	Default value
byte	0
short	0
int	0
long	0
float	0.0
double	0.0
char	'\u0000'
boolean	false
String	null

7.2 Array operations

- Array elements can be accessed using index.
- Array index starts with 0.

Example

```
int[ ] myArray = new int[ 5 ];

System.out.println( myArray[ 0 ] ); //0
System.out.println( myArray[ 1 ] ); //0
System.out.println( myArray[ 2 ] ); //0
System.out.println( myArray[ 3 ] ); //0
System.out.println( myArray[ 4 ] ); //0

System.out.println( myArray[ 5 ] ); // ArrayIndexOutOfBoundsException
```

```
// Filling array

myArray[ 0 ] = 10;
myArray[ 1 ] = 20;
myArray[ 2 ] = 30;
myArray[ 3 ] = 40;
myArray[ 4 ] = 50;

for ( int i = 0 ; i < myArray.length ; i++ ) {

    System.out.println( myArray[ i ] );

}
```

Questions

Write code to fill an array with 10 natural numbers using loop

Write a code to print the following array using loop

String[] superheroes = { "Batman", "Ironman", "Thor", "Superman", "Spiderman" };

Write code to print sum of all elements in the following array

```
int[ ] myArray = { 1, 3, 4, 7, 8, 10 };
```

Write code to print maximum of all elements in the following array

```
int[ ] myArray = { 1, 9, 4, 8, 2, 3 };
```


Write code to copy the following array to another

```
int[ ] myArray = { 1, 9, 4, 8, 2, 3 };
```

7.3 Multi-dimensional array

- A two-dimensional array is basically an array at every index of parent array.
- Elements have a **row** and a **column** index.
- For operations on two-dimensional array, we must use **nested for** loops.

Example

```
int[ ][ ] matrix = new int [ 2 ][ 3 ];
```

or

```
int[ ][ ] matrix = {  
    { 1, 2, 4},  
    { 3, 6 ,9}  
};
```

```
System.out.println( matrix[ 0 ] );
```

```
// It will return { 1, 2, 4 } - (array inside array)
```

```
System.out.println( matrix[ 0 ][ 0 ] );
```

```
// It will return 1 - first element of first array or element at first row, first column
```

```
for ( int i = 0 ; i < matrix.length ; i ++ ) {
```

```
    for ( int j = 0 ; j < matrix[ i ].length ; j++ ) {
```

```
        System.out.print ( matrix [ i ][ j ] );
```

```
    }
```

```
System.out.println( );
```

```
}
```

7.4 Enhanced for loops

- Enhanced for loops were introduced in Java 5.
- Enhanced for loops are used to iterate over arrays and collections.

Example

```
for ( elementReference : arrayName ) {  
  
    System.out.println ( elementReference );  
  
}  
  
int[ ] myArray = { 1, 2, 3, 4, 5 };  
  
for ( int element : myArray ) {  
  
    System.out.println( element );  
  
}
```

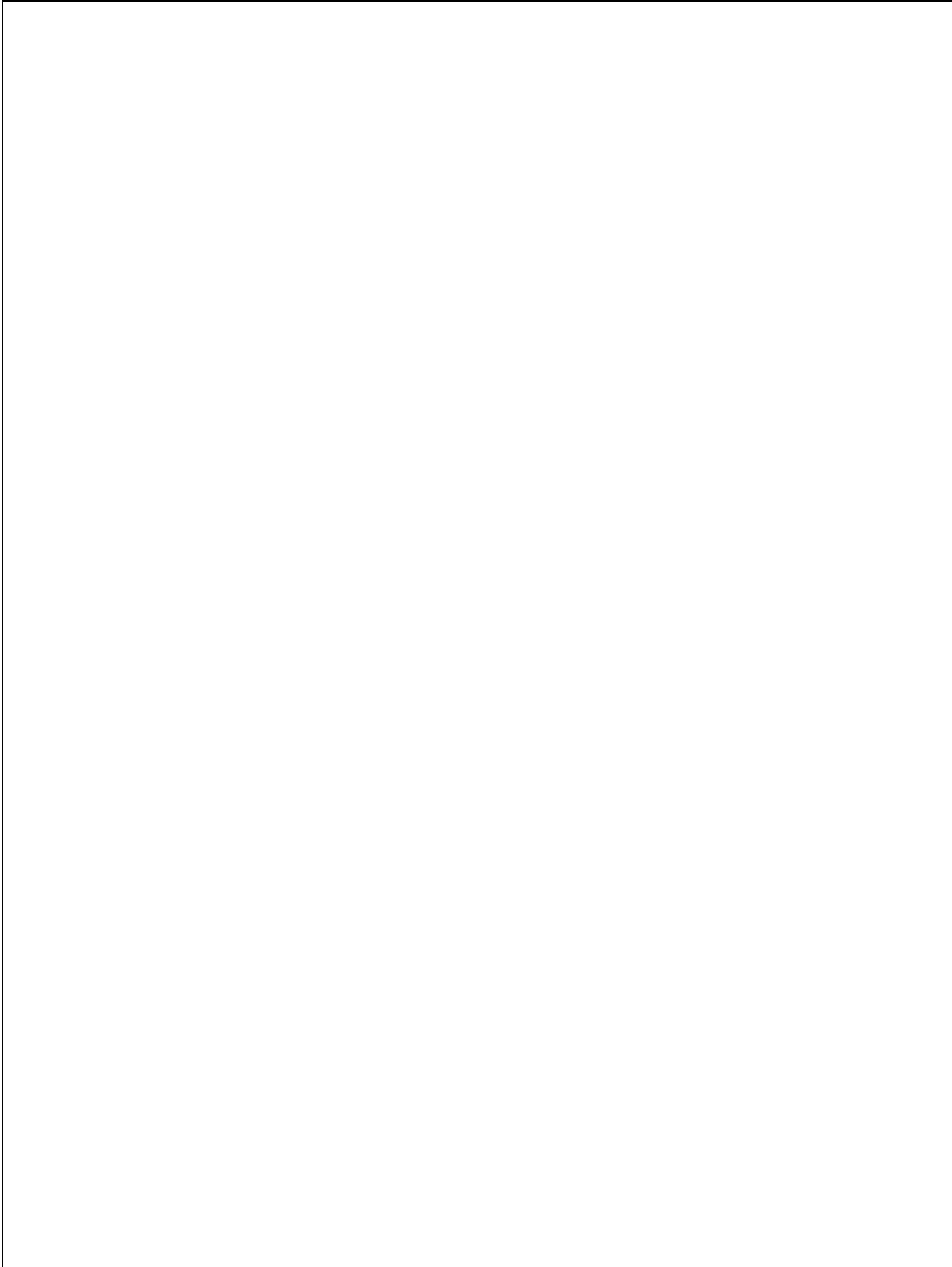
Questions

Write code to print the following arrays using enhanced for loop-

```
int[ ] myArray = { 1, 9, 4, 8, 2, 3 };
```

```
String[ ] superheroes = { "Batman", "Ironman", "Thor", "Superman", "Spiderman" };
```

```
charArray = { 'a' , 'b' , '$' , '2' };
```



7.5 Variable number of arguments

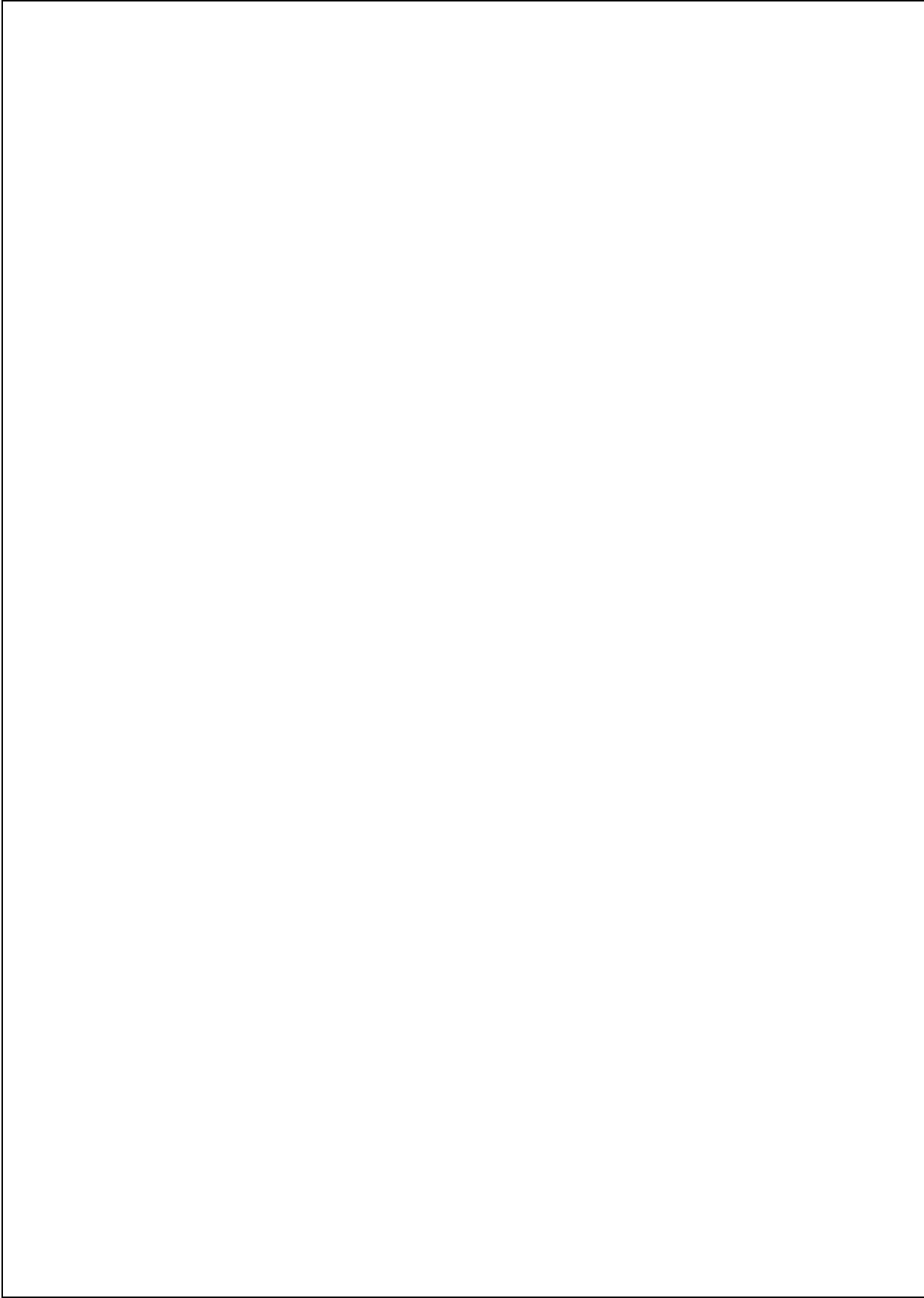
- The concept of variable number of arguments allows us to pass any number of arguments every time we call the same method.
- Instead of overloading method, now we can create just a single method to add 2 or 3 or 4 numbers.
- Variable number of arguments is actually like passing an array as an argument.

Example

```
public static int sum ( int . . . numbers ) {  
  
    int result = 0;  
  
    for( int num : numbers ) {  
  
        result += num;  
    }  
  
    return result;  
}  
  
// Calling method  
  
sum ( 2, 3 );  
sum ( 3, 4, 5 );  
sum ( 1, 2, 6, 7 );
```

Questions

Write the previous Calculator class, replace the all basic and overloaded methods with single method with variable number of arguments.



8. Object Oriented Programming

8.1 Classes and Objects

- **Objects** in general terms is any real world entity that possess some properties and some behaviours.
- **Property** defines attribute, characteristics or quality of an object.
- **Behaviour** defines operations an object can perform.
- Example of objects - car, human, chair, laptop etc.

Let's take an example of car.

Properties of car

- Brand
- Color
- Top speed
- Length
- Height

Behaviours of car

- Start
- Move
- Stop

Let's try to figure out data type of each property of car

Property	Datatype
Brand	String
Color	String
Top speed	double
Length	double
Height	double

Let's try to convert car object into code

Class - class is a blueprint of an object that defines properties and behaviour of an object.

```
class Car {  
  
    String brand;  
    String color;  
    double topSpeed;  
    double height;  
    double length;  
  
    public void start(){  
  
    }  
  
    public void move(){  
  
    }  
  
    public void stop(){  
  
    }  
}
```


Questions

List 10 objects you see in real world

Choose one object from your list and mention at least 5 properties and 5 behaviours.

Properties	Behaviours

Choose the appropriate data type for the properties that you have mentioned

Property	Datatype

Create a class for the object that you have chosen.

--

Creating objects in Java

- Objects are created using **new** keyword.
- When an object is created, memory space is allocated to all the members of the class and all the class variables are initialized with default value.
- Object creation process consist of three steps - **declaration**, **instantiation** and **initialization**

```
// Creating car object using new keyword.
```

```
new Car( );
```

```
// In this step an Car object is created without reference, this type of object is known as an orphan object. This object cannot accessed for further use.
```

Declaration

```
Car audi;
```

Instantiation and initialization

```
audi = new Car();
```

```
// new keyword is used to instantiate (Allocating memory to all members of class)
```

```
// Car() is used to initialize (Initializing member variables with default values)
```

Instantiation (Allocating memory to all members)

brand
color
topSpeed
height
length
start()
move()
stop()

Initialization (Initializing member variables or class variables with default value)

null
null
0.0
0.0
0.0
start()
move()
stop()

Hence new definition of object is -

Object - is an instance of a class.

Accessing members of object using **dot** operator

```
Car audi = new Car( );

audi.brand = "Audi";
audi.color = "Black";
audi.topSpeed = 200;

System.out.println (audi.brand)
System.out.println (audi.color)
System.out.println (audi.topSpeed)

audi.start( )
audi.move( )
audi.stop( )
```

Questions

Create at least 2 objects of the class that you have created in previous question. Access the member variables and change their values

8.2 Constructors

- **Constructors** are special methods that are used to **initialize** the member variables of a class with default values.
- Constructor name is always **same as class name**.
- Constructors do not return any value and therefore **do not have return types**.

Implicit constructor

Even if we don't create a constructor, JVM provides a default constructor called the **implicit constructor**. Implicit constructor is responsible for initializing all the class variables with default values.

Explicit constructors

If we wish to have our own default values rather than the one provided by JVM, then we must create a constructor explicitly known as **explicit constructor**.

```
class Car {  
  
    String brand;  
    String color;  
    double topSpeed;  
    double height;  
    double length;  
  
    Car ( ) {  
  
        color = "white";  
        topSpeed = -1;  
        height = -1;  
        length = -1;  
    }  
  
}
```

Question

Consider above example, what default value does **brand** gets

Parameterized constructor

If we wish to assign values to class members at the time of object creation, then we must create constructors with parameters known as **parameterized constructors**.

```
class Car {  
  
    String brand;  
    String color;  
    double topSpeed;  
    double height;  
    double length;  
  
    Car (String brand,String color, double topSpeed, double height,  
        double length ) {  
  
        this.brand = brand;  
        this.color = color;  
        this.topSpeed = topSpeed;  
        this.height = height;  
        this.length = length;  
    }  
  
}  
  
//Creating object  
  
Car audi = new Car( "Audi", "Black", 200, 70, 150 );
```

this keyword is used to refer to the current object. It is used to avoid confusion (ambiguity) between class variables and parameters (Since they have same name).

Questions

Create parameterized constructor for the class that you have created in previous question

Constructor overloading

- Just like method overloading, a class can have multiple constructors with different number of argument, or different type of argument or different order of argument.
- A constructor can call another constructor using **this** keyword.

```
class Car {  
  
    String brand;  
    String color;  
    double topSpeed;  
    double height;  
    double length;  
  
    Car ( ) {  
  
        color = "white";  
        topSpeed = -1;  
        height = -1;  
        length = -1;  
    }  
  
    Car (String brand,String color, double topSpeed) {  
  
        this( );  
        this.brand = brand;  
        this.color = color;  
        this.topSpeed = topSpeed;  
  
    }  
  
}
```

8.3 Inheritance

- The concept of inheriting properties and behaviours from parent to child is known as **inheritance**. Parent or child can be a class, interface or abstract class.
- In this chapter we will learn inheritance only with classes. In the next chapter we will learn inheritance with interface and abstract class.
- A class inherits properties and behaviours from another class using **extend** keyword.

```
class Vehicle {  
  
    String brand = "BMW";  
  
    public void start() {  
  
        System.out.println( "Vehicle started" );  
  
    }  
  
}  
  
class Car extends Vehicle {  
  
}  
  
class Test {  
  
    public static void main( String[ ] args ) {  
  
        Car carObj = new Car();  
  
        System.out.println( carObj.name );  
  
        carObj.start();  
  
    }  
  
}
```

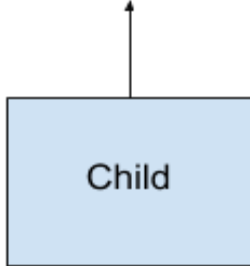
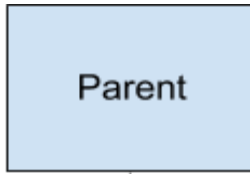
Method overriding

- If child class wish to change the content of method inherited from parent class, this can be achieved by creating a method in child class with same name but different content.
- The concept of having method with same name as in parent class, but different content is known as **method overriding**.
- A child can access parent's properties or behaviours using **super** keyword.

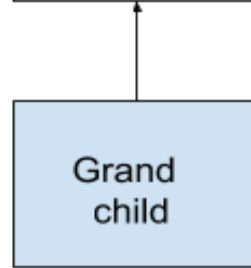
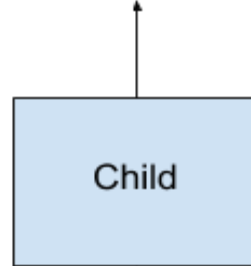
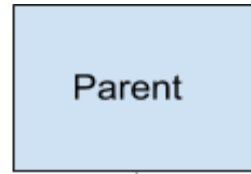
```
class Car extends Vehicle {  
  
    public void start (){  
        System.out.println( "Car started" );  
    }  
  
    public void displayBrand(){  
        System.out.println( super.brand );  
    }  
}
```

Types of inheritance

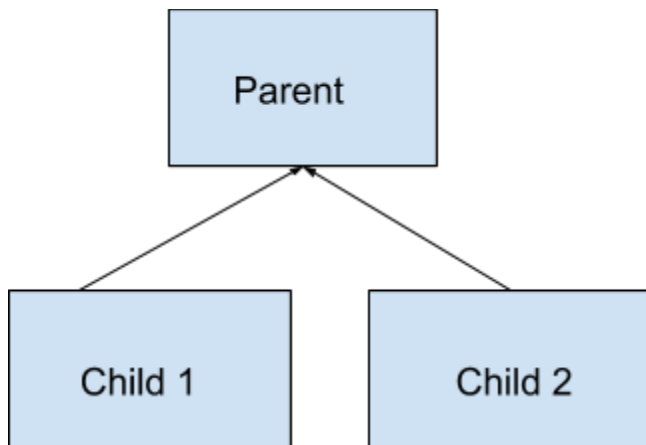
- Single inheritance
- Multi-level inheritance
- Hierarchical inheritance
- Multiple inheritance



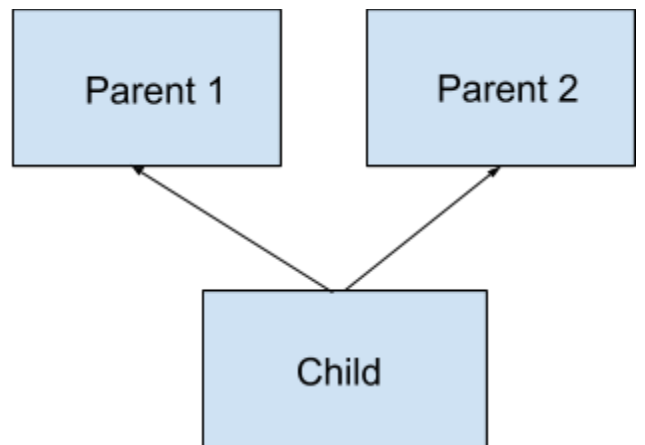
Single



Multi-level



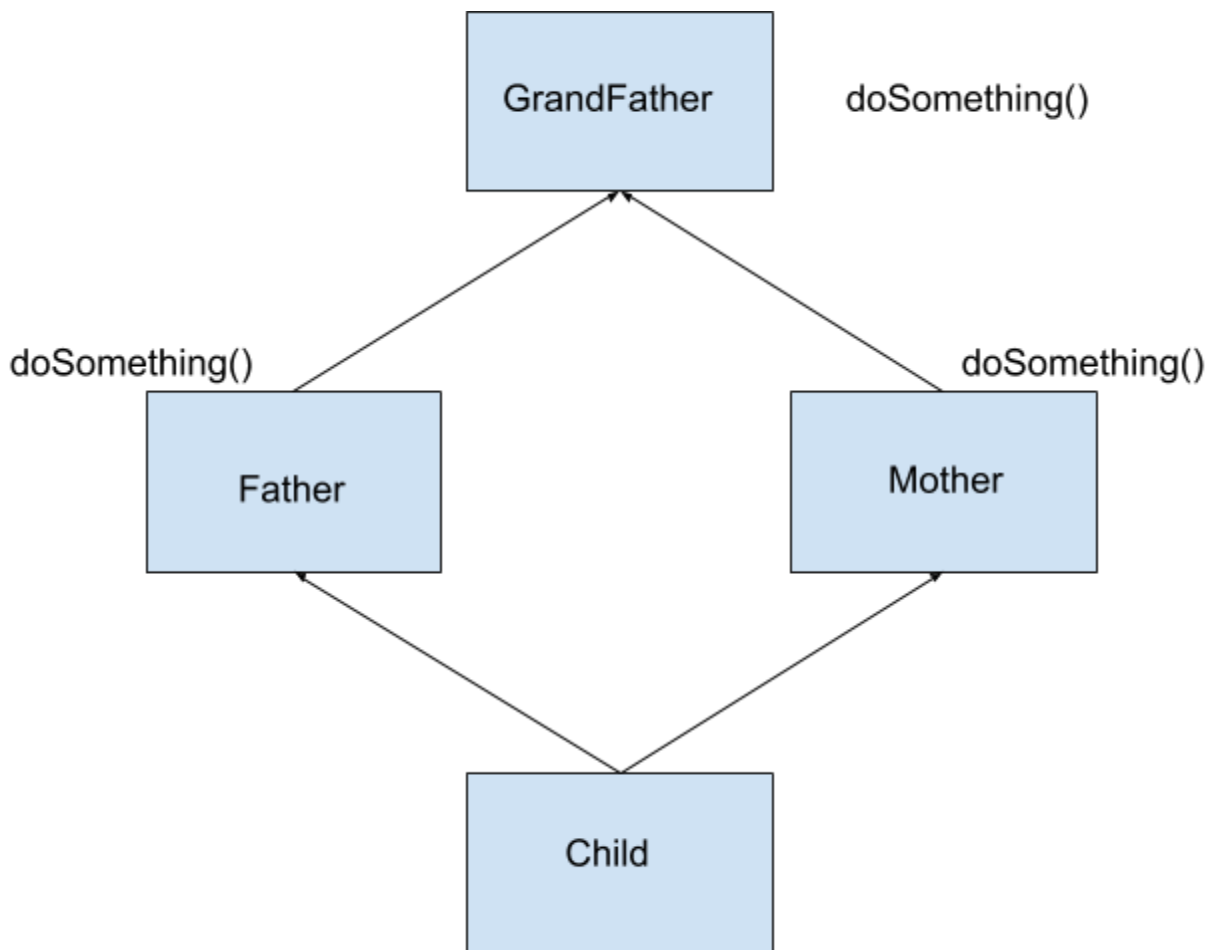
Hierarchical



Multiple

Inheritance	Example
Single	<pre> class A{ } class B extends A { } </pre>
Multi-level	<pre> class A { } class B extends A { } class C extends B { } </pre>
Hierarchical	<pre> class A { } class B extends A { } class C extends A { } </pre>
Multiple	Not possible with classes

Why multiple inheritance is not possible with classes



The above diagram depicts why multiple inheritance is not possible with classes. Consider a **GrandFather** class with **doSomething()** method. **Father** and **Mother** class inherits from **GrandFather** class and **override** the **doSomething()** method. But when **Child** class inherits from **Father** and **Mother** class, there is a **confusion** whether to override **doSomething()** method from **Father** class or **Mother** class.

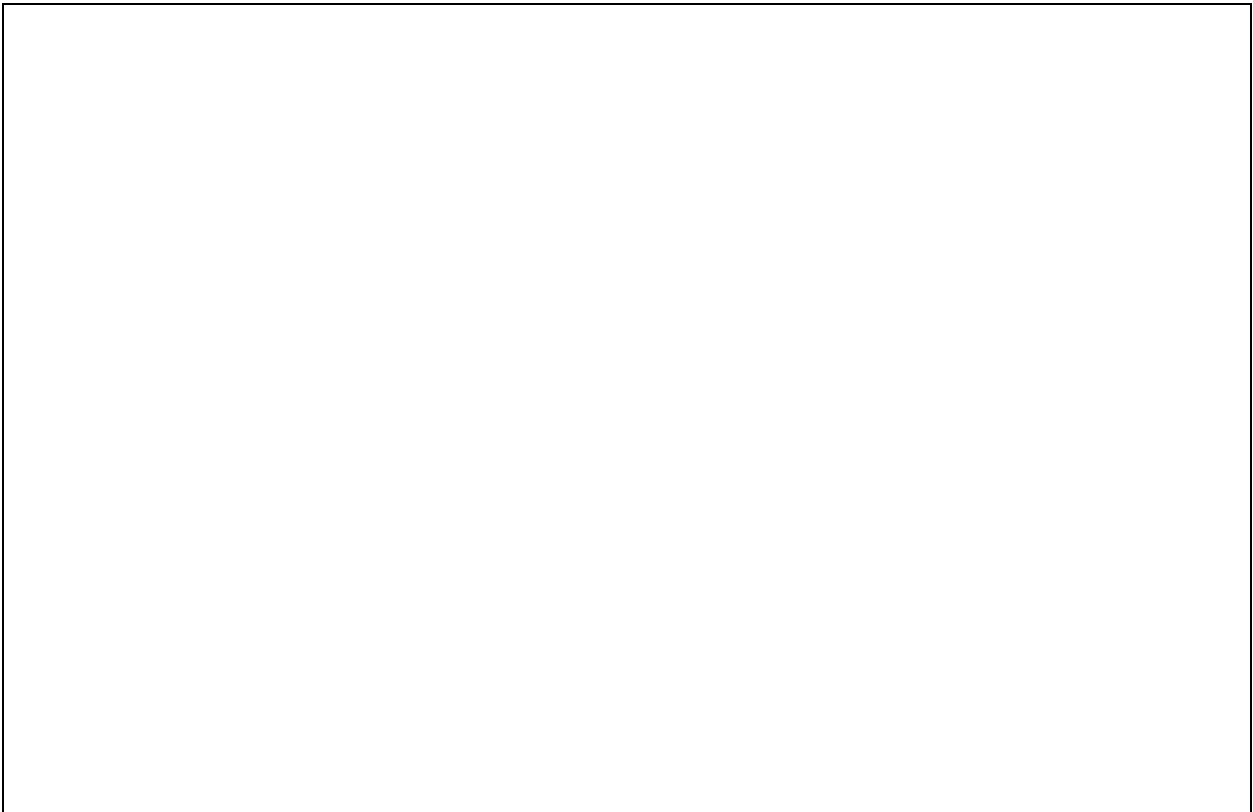
Due to this **ambiguity**, multiple inheritance is not possible with classes.

Questions

Guess the output

```
class A {  
  
    public void introduce( ) {  
  
        System.out.println( "I am class A" );  
  
    }  
  
    public void greet( ){  
  
        System.out.println("Hello from class A");  
  
    }  
  
}  
  
class B extends A {  
  
    public void introduce() {  
  
        super.introduce( );  
        System.out.println( " I am class B" );  
    }  
  
}  
  
class C extends B {  
  
}
```

```
class Test {  
    public static void main ( String[ ] args ) {  
        C c = new C( );  
        c.introduce( );  
        c.greet( );  
    }  
}
```



8.4 Object class

Consider the following code

```
class Student {  
    int id;  
    String name;  
  
    Student(int id, String name){  
        this.id=id;  
        this.name=name;  
    }  
}  
  
class Test {  
    public static void main(String[] args) {  
        Student s1 = new Student(1,"Sachin");  
        Student s2 = new Student(2,"Ponting");  
  
        System.out.println(s1.id);  
        System.out.println(s1.name);  
  
        System.out.println(s2.id);  
        System.out.println(s2.name);  
    }  
}
```

We know the output of the above program very well, ever wondered what if we try to print the object reference itself.

```
System.out.println(s1);  
  
System.out.println(s2);
```

Output -

```
com.itv.demo.Student@15db9742  
  
com.itv.demo.Student@6d06d69c
```

Output show class name along with package name concatenated with @ and some sequence of characters

Let's see another example

```
System.out.println(s1.toString());  
  
System.out.println(s2.toString());
```

Output -

```
com.itv.demo.Student@15db9742  
  
com.itv.demo.Student@6d06d69c
```

- Output is exactly same. But we never created any **toString()** method in our class.
- The toString() method comes from **Object** class.
- Object class is the **super class of all class**.
- All classes **extends Object class by default**.

Let's override **toString()** method to get meaningful output

```
class Student {  
  
    int id;  
    String name;  
  
    Student(int id, String name){  
  
        this.id=id;  
        this.name=name;  
  
    }  
  
    @Override  
    public String toString() {  
  
        return "Student [id=" + id + ", name=" + name + "]";  
  
    }  
}  
  
class Test {  
  
    public static void main(String[] args) {  
  
        Student s1 = new Student(1,"Sachin");  
  
        Student s2 = new Student(2,"Ponting");  
  
        System.out.println(s1);  
        System.out.println(s2);  
  
    }  
}
```

Output-

Student [id=1, name=Sachin]

Student [id=2, name=Ponting]

8.5 Abstraction

- In general terms, abstraction is like telling what to do, but not how to do.
- **Abstraction** is a process of providing functionality without implementation details.
- Abstraction is achieved with the help of **interface** and **abstract class**.

Interface

- Interface is like a class, but it consist of **abstract methods**.
- **Abstract method** is a method without body.
- Interface do not have constructors.
- Object of interface cannot be created.
- Purpose of creating an interface is to create a highly generalized blueprint that can be inherited by more specialized classes.
- A class can inherit from an interface using **implements** keyword.
- Once a class implements an interface, it becomes compulsory to override the the abstract methods of interface.

```
interface Flyable {  
  
    public void fly ( );  
  
}  
  
class Aeroplane implements Flyable {  
  
    public void fly(){  
  
        System.out.println( "Run and fly" );  
  
    }  
  
}
```

```
class Helicopter implements Flyable {  
    public void fly() {  
        System.out.println( "Stand and fly" );  
    }  
}
```

Multiple inheritance is possible with interfaces. A class can implement more than one interface.

```
class Aeroplane implements Flyable, Runnable {  
  
}
```

Abstract class

- An **abstract class** consist of both abstract as well as concrete methods.
- Since methods in abstract classes are not abstract by default, a method is declared with **abstract** keyword.
- Abstract class can have constructors.
- Object of abstract class cannot be created.
- A class inherits from an abstract class using **extend** keyword.

```
abstract class Flyable {  
  
    abstract public void fly ();  
  
    public void firstAid( ){  
  
        }  
  
}
```

Important - In Java 8.0 many new features were introduced, one of them was that, interface can have concrete methods as well. Interface can have **default** and **static** methods as concrete methods.

Base class can override a default method(not compulsory), but cannot override a static method.

```
interface Flyable {  
  
    default void firstAid( ){  
  
        }  
  
    static void communicate( ){  
  
        }  
  
}
```

Questions

State true or false.

Statement	Answer
class extends class	
class extends interface	
class implements interface	
class implements interface1, interface2	
class extends class1, class2	
class implements class1, class2	
class extends class implements interface	
Class extends class implements interface1, interface2	
interface implements interface	
Interface implements interface1, interface2	
interface extends interface	True
Interface extends interface1, interface2	True

8.6 Packages

- **Package** is a folder in Java, where all related classes are placed together.
- Just like we keep our songs organized in folders depending upon genre, artist, album etc, similarly packages are created to keep classes organized.
- A package is created using **package** keyword.

```
package A;
```

```
class Student {  
  
}
```

- Let's create object of student class in different package.
- In order to access student class from different package, student class must have **public** access modifier.
- We will learn about access modifiers in detail in next chapter.

```
package A;
```

```
public class Student {  
  
}
```

```
-
```

```
package B;
```

```
class Test {  
  
    Public static void main(String[ ] args) {  
  
        A.Student s1 = new A.Student( );  
        A.Student s2 = new A.Student( );  
  
    }  
}
```



```
}
```

Instead of using package name as reference every time, we can use **import** keyword.

```
package B;

import A.Student;

class Test {

    Public static void main(String[ ] args) {

        Student s1 = new Student( );
        Student s2 = new Student( );

    }

}
```

Packages should be named correctly. We will learn about industrial naming conventions in upcoming chapters. Packages should be named in the following manner -

Organization type. Organization name. Package name

Example -

```
com.itvedant.admin
org.itvedant.student
```

To import all the classes from a package, we code -

```
import com.itvedant.admin.*;
```

** is used to represent all the classes of that particular package.*

*But we should remember, * imports all the classes of the current package but not of the sub-packages.*

```
package com.itvedent.student;

public class Student {

}

-----
-

package com.itvedant.admin;

import com.itvedant.student.Student;

class Test {

    Public static void main(String[ ] args) {

        Student s1 = new Student( );

    }

}
```

Important - Java by default imports **java.lang** package that contains classes and interfaces that are essential to writing java program, example - **System, String, Thread** etc.

We can create two classes with same name if they belong different packages.

Hence, we can create our String class without any error.

8.7 Access Modifiers

- Access modifiers specify the levels of access for a class, a variable and a method.
- We understand accessing a variable or a method.
- Accessing a class means, creating object of that class or inheriting that class.

Access modifier	Within class	Within package	Outside package (Universal)
default	yes	yes	no
public	yes	yes	yes
protected	yes	yes	Only with inheritance
private	yes	no	no

Apart from the above access modifiers, there are many other important modifiers -

- **static**
- **final**
- **abstract**
- **synchronized**
- **transient**
- **native**
- **strictfp**

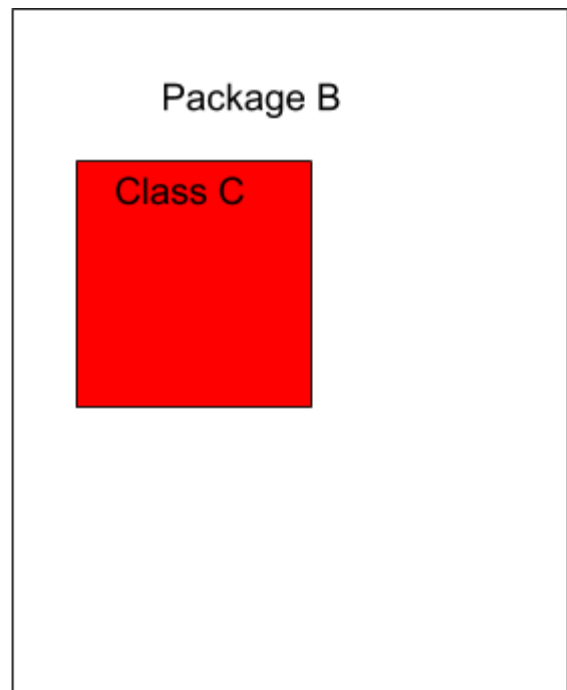
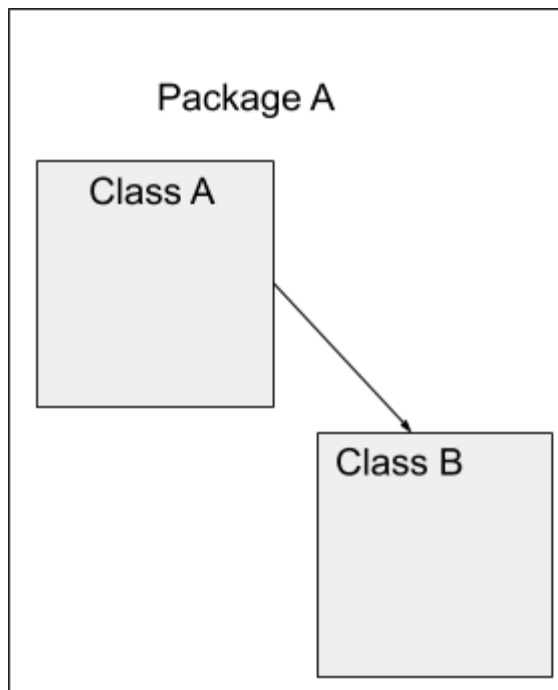
We will learn **synchronized** and **transient** in the upcoming classes.

The remaining modifiers - **native** and **strictfp** is left to you. Do some research and get back to your trainer :)

Default

- If a class or a method or a variable do not have any access modifier, that means they can be accessed within the package.
- No need to mention **default** keyword.

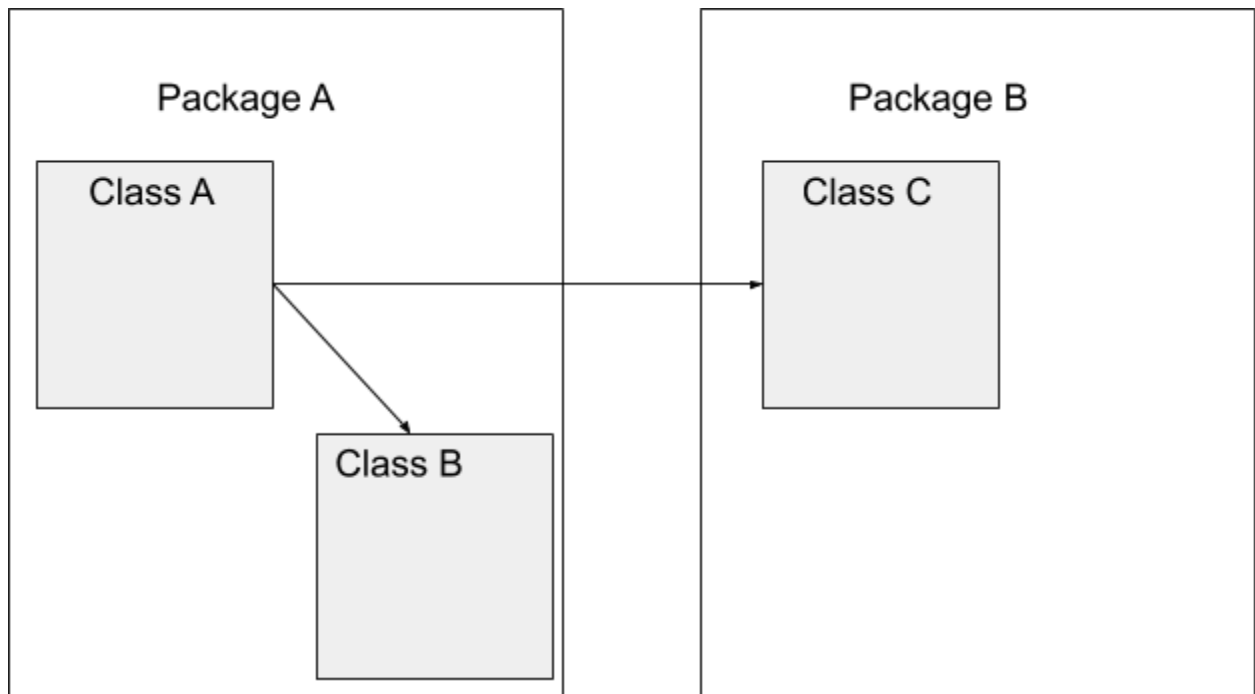
```
class Test {  
    int var;  
    void doSomething( ) {  
    }  
}
```



Public

- If a class or a method or a variable have **public** access modifier, that means they can be accessed outside the package (universal).
- Need to mention **public** keyword.

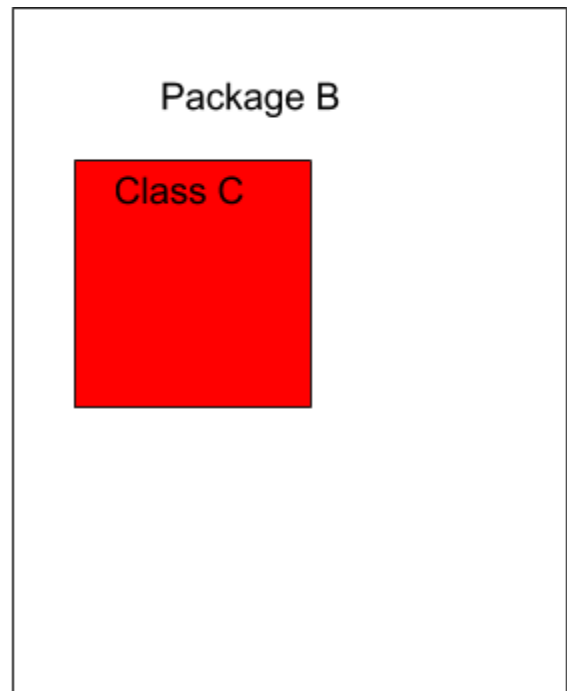
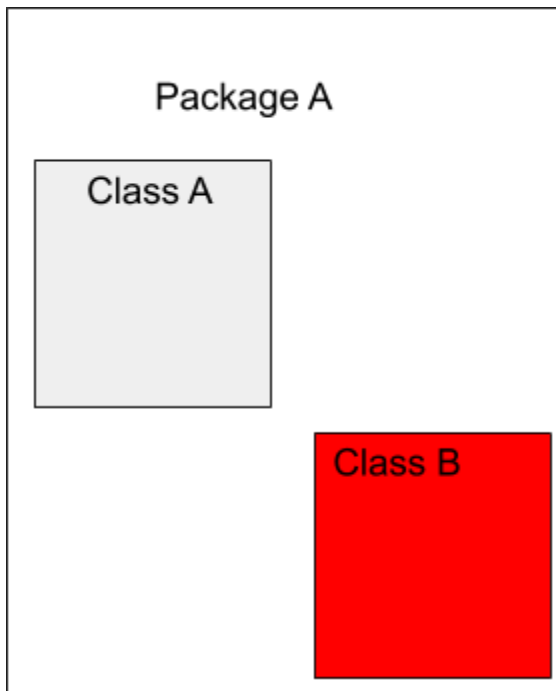
```
public class Test {  
  
    public int var;  
  
    public void doSomething( ) {  
  
    }  
  
}
```



Private

- If a method or a variable have **private** access modifier, that means they can be accessed only within the class.
- **A class cannot be private.**
- Need to mention **private** keyword.

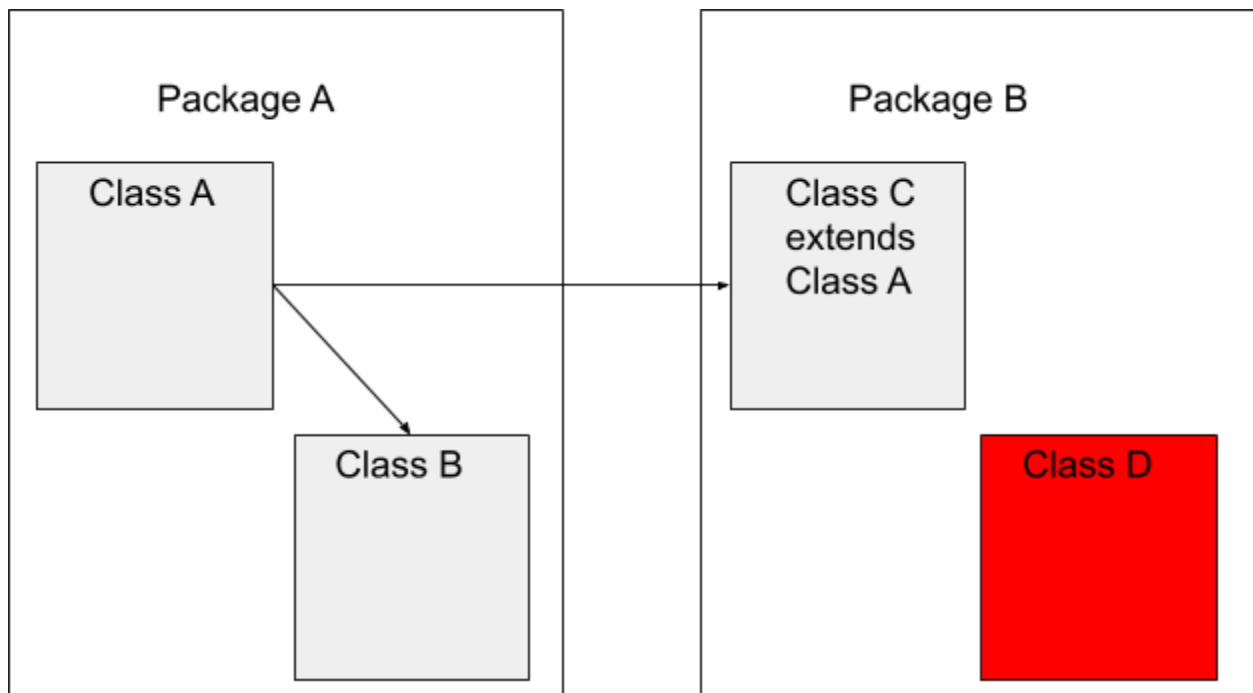
```
class Test {  
  
    private int var;  
  
    private void doSomething( ) {  
  
    }  
  
}
```



Protected

- If a method or a variable have **protected** access modifier, that means they can be easily accessed within the package but only **subclasses** can access them outside the package (Inheritance).
- **A class cannot be protected.**
- Need to mention **protected** keyword.

```
class Test {  
  
    protected int var;  
  
    protected void doSomething( ) {  
  
    }  
  
}
```



Final

- A **variable** declared as final becomes a **constant**, it's value can't be changed in future.
- A **method** declared as final cannot be **overridden** by subclasses.
- A **class** declared as final cannot be **inherited** by other classes.
- Need to use **final** keyword.

```
final class Test {  
  
}  
  
final double PI = 3.14;  
  
public final void doSomething( ) {  
  
}
```

Static

- A **variable** declared as static will have a common value for all the instance (objects) of the class. We can also say that static variable have a single copy in the memory.
- A **method** declared as static will also be having a single copy in the memory.
- Unlike the member variables (class variables) or member methods of class, **static members** can be accessed directly with **class reference**, instead of **object reference**.
- **A class cannot be static.**
- Need to use **static** keyword.

```
class Test {  
  
    static int var;  
  
    public static void doSomething( ) {  
  
    }
```



```
}
```

Abstract

- A **method** declared as abstract do not have body.
- A **class** is declared as abstract when it has abstract methods.
- **A variable cannot be abstract.**
- Need to use **abstract** keyword.

```
abstract class Flyable {  
  
    abstract public void fly ();  
  
}
```

Questions

Explain all the access modifiers with respect to class, variable and method.

	Class	Variable	Method
default			
public			

protected	NA		
private	NA		
final			
static	NA		
abstract		NA	

Important - Interface methods are **abstract** and **public** by default and interface variables are **final** and **static** by default

8.8 Industrial Naming Convention

Class	<i>Class name should start with capital letter, and first letter of every consequent word should be capital</i> <i>Class name should be a noun</i>	Student, Test, DatabaseManager, NetworkUtility
Interface	<i>Same convention as class name</i> <i>Interface name should be an adjective</i>	Flyable, Runnable, Serializable
Variable and method	<i>Variable name should start with small letter, and first letter of every consequent word should be capital</i> <i>Variable name should be a noun</i>	name, radiusOfCircle, areaOfSquare
Final variable	<i>Final variable name should be complete capitals, every consequent word should be separated with _</i>	PI_VALUE, DB_URL
Package	<i>Organization type . organization name . package name</i>	com.itvedant.admin, org.itvedant.student

Questions

Give 5 examples each with appropriate industrial naming convention

Class	
Interface	

Variable	
Final variable	
Package	

8.9 Encapsulation

- **Encapsulation** is a concept of binding the data and function acting on the data, together in single unit.
- One of the advantage of encapsulation is **Data hiding**.
- Direct access of data can be avoided by declaring them with **private** access modifier and indirect access can be given using **public getter and setters** method.

```
class Student {

    private String name;

    private int id;

    public void setName ( String name ) {

        this.name = name;

    }

    public String getName( ){

        return this.name;

    }

    public void setID ( int id ){

        this.id = id;

    }

    public int getID( ){

        return this.id;

    }

}
```

```
}
```

Questions

Implement data hiding in the following code

```
class Employee {
```

```
    _____ String name;
```

```
    _____ double salary;
```

```
    _____ int id
```

```
}
```

8.10 Polymorphism

- Poly means many and morph means forms.
- Hence polymorphism means ability of an object to take more than one form.

```
interface Vehicle {  
    void move( );  
}  
  
class Car implements Vehicle {  
    public void move( ){  
        System.out.println( "Car is moving" );  
    }  
}  
  
class Truck implements Vehicle {  
    public void move( ){  
        System.out.println( "Truck is moving" );  
    }  
}  
  
class Test {  
    public static void main(String[ ] args ) {  
        Vehicle v;
```

```
        v = new Car( );  
        v.move( );  
  
        v = new Truck( );  
        v.move( );  
  
    }  
}
```

Above example shows that object of type **Vehicle** acts differently depending on the code. When object of type Vehicle is initialized with constructor of **Car**, it gives a different output than when initialized with constructor of **Truck**.

There are two types of polymorphism in Java

- 1) Run time polymorphism
- 2) Compile time polymorphism

Run time polymorphism

- Example of run time polymorphism is **method overriding**.
- The concept of having method with same name as in parent class, but different content is known as method overriding.
- Method overriding is resolved by **JVM at run time**, hence it is known as run time polymorphism.

Give example of run time polymorphism

Compile time polymorphism

- Example of compile time polymorphism is **method overloading**.
- Method overloading is a concept where in a class we can create multiple methods with same name, but they must have different number of parameters or different types of parameters or different order of parameters.
- Method overloading is resolved by **Compiler** at **compile time**, hence it is known as compile time polymorphism.

Give example of compile time polymorphism

9. Nested Classes

- A class created inside another class is known as **nested or inner class**.
- It is a way of grouping classes that are used only at one place.
- Using nested classes increases encapsulation and creates more readable and maintainable code.

Important - Java treats inner classes as members of class, hence inner classes can be **private, protected and static**.

Types of nested class

- 1) Member inner class
- 2) Static inner class
- 3) Local inner class
- 4) Anonymous inner class
- 5) Lambda expression

9.1 Member inner class

- A member inner class is like any other non-static member of class.
- Just like any member (non-static) of class, member inner class is also accessed with the object of outer class.
- Member inner class can access the **private** members of the class
- **Static** methods cannot be declared inside a member inner class.

```
class Outer {  
    private int i;  
    class Inner {  
        public void doSomething( ){  
            System.out.println( i );  
        }  
    }  
}  
//Creating object  
class Test {  
    public static void main( String[ ] args ) {  
        Outer out = new Outer( );  
        Outer.Inner in = out.new Inner( );  
        //or  
        Outer.Inner in = new Outer.new Inner( );  
    }  
}
```

9.2 Static inner class

- A static inner class is like any other static member of class (Common for all instance).
- Just like any other static member of class, static inner class is also accessed with the outer class reference.
- Static inner class can access only the **static** members of class (can be **private**).
- **Static** methods can be declared inside a static inner class.

```
class Outer {  
  
    private static int i;  
  
    static class Inner {  
  
        public void doSomething( ){  
  
            System.out.println( i );  
  
        }  
  
    }  
  
}  
//Creating object  
  
class Test {  
  
    public static void main( String[ ] args ) {  
  
        Outer.Inner in = new Outer.Inner( );  
  
    }  
  
}
```

9.3 Local inner class

- A variable inside a method is known as **local variable**.
- Similarly a class inside a method is called **local inner class**.
- Local inner class can be instantiated only within the method.
- Local inner class can access **local** variables, **static** members and **non-static** members (including **private**).

```
class Outer {  
    public void doSomething( ) {  
  
        class Local {  
  
        }  
  
        Local local = new Local( );  
    }  
}
```

9.4 Anonymous Inner class

- Anonymous inner class is an inner class without any name.
- It is like implementing an interface or extending a class (abstract or concrete) without having to actually subclass a class.

```
interface Flyable {  
  
    void fly( );  
  
    void land( );  
  
}  
  
class Test {  
  
    public static void main ( String[ ] args ){  
  
        Flyable obj = new Flyable ( ){  
  
            public void fly( ){  
  
                }  
  
            public void land( ){  
  
                }  
  
        };  
  
    }  
  
}
```

9.5 Lambda Expression

- Lambda expression is like an anonymous inner class of a **functional interface**.
- Functional interface is an interface that has only one abstract method.
- A functional interface is created by annotating an interface as **@FunctionalInterface**.
- One of the useful functional interface we will come across in upcoming chapters is **Runnable**.

```
@FunctionalInterface
```

```
interface Flyable {
```

```
    void fly( );
```

```
}
```

```
class Test {
```

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

```
        Flyable obj = ( ) -> {
```

```
            System.out.println( "Hello world" );
```

```
        };
```

```
    }
```

```
}
```

Questions

Create anonymous inner class for the following interface

```
interface Playable {  
    void play( );  
    void pause( );  
    void stop( );  
}
```


Create a functional interface of your choice and create lambda expression for the same

10. String

- String is defined as an array of characters.
- String is a class in Java.
- String objects can be created in two ways
 - 1) Using string literal
 - 2) Using new keyword

```
// Using string literal
```

```
String s1 = "Batman";  
String s2 = "Superman";
```

```
// Using new keyword
```

```
String s1 = new String( "Batman" );  
String s2 = new String( "Superman" );
```

```
String name = "Batman";
```

```
// Is equivalent to
```

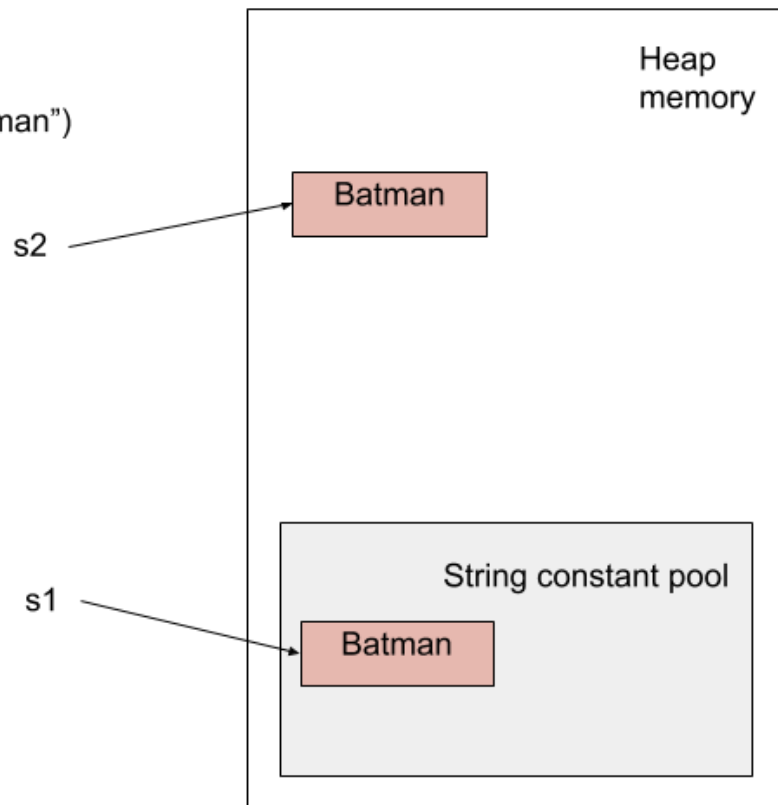
```
char[ ] name = { 'B', 'a', 't', 'm', 'a', 'n' };
```

10.1 String memory management

- A string created using new keyword is stored in **heap memory**.
- A string created using literals is stored in **string constant pool**.
- String constant pool lies inside heap memory.

String s1 = "Batman"

String s2 = new String("Batman")



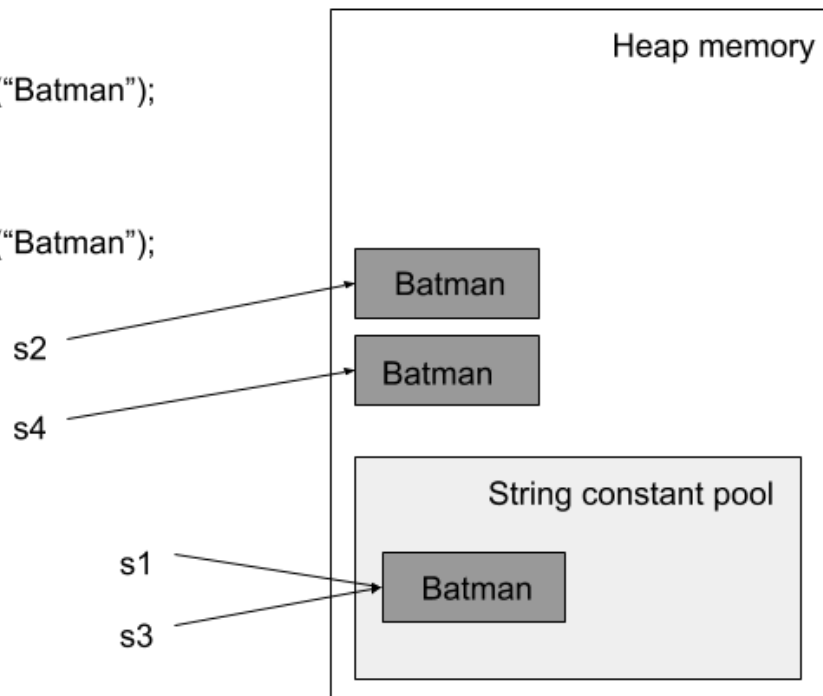
- Strings created using **new** keyword, always creates a new string object in the **heap memory**.
- When a string is created using string **literals**, **jvm** checks if the string object is already present in the **string constant pool** or not. If the object is not present, then **jvm** creates a **new object**, else if the object is already present, jvm creates a **new reference** to the previous object.

String s1 = "Batman";

String s2 = new String("Batman");

String s3 = "Batman";

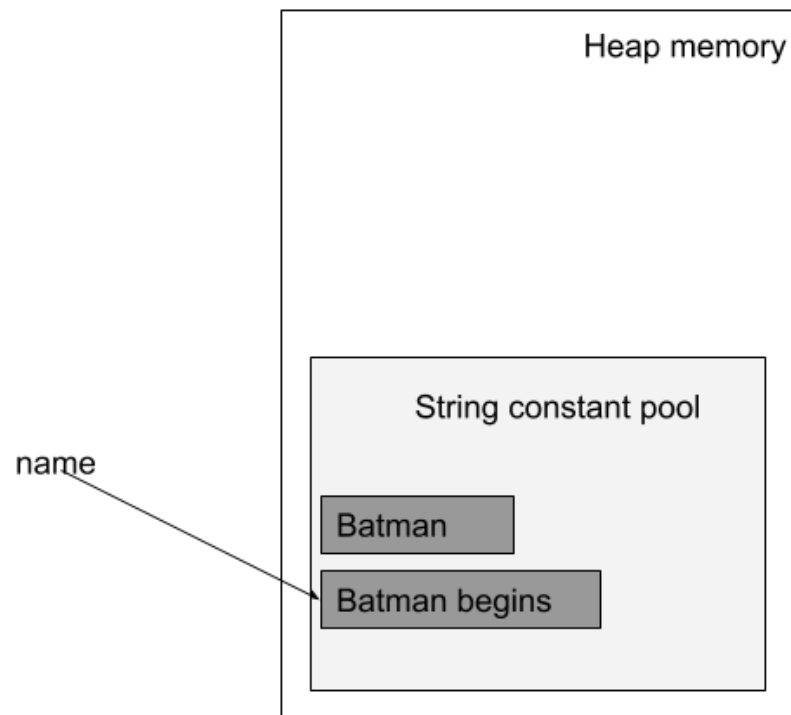
String s4 = new String("Batman");



10.2 Immutable property of string

- **Immutable property of string** defines that once a string object is created, that object cannot be changed in future.
- If you try to change a string object, it creates a new object instead of making any change to the current object.
- The reference to the current object shifts to the new object.
- The previous object becomes an orphan object and becomes eligible for garbage collection.

```
String name = "Batman";  
  
name = "Batman returns";
```



10.3 == operator vs equals() method

- == operator checks if two strings objects refer to same object or not.
- **equals()** method belongs to **Object class**.
- **String class overrides equals()** method that checks if two string objects have same value or not.

```
String s1 = "ITVedant";  
  
String s2 = new String("ITVedant");  
  
String s3 = "ITVedant";  
  
String s4 = "FieryDevs";  
  
s1 == s2;  
  
// Returns false  
  
s1.equals(s2);  
  
// Returns true  
  
s1 == s3;  
  
// Returns true  
  
s1.equals(s3);  
  
// Returns true  
  
s1 == s4  
  
// Returns false  
  
s1.equals(s4);  
  
// Returns false
```

10.4 String methods

The url <https://docs.oracle.com/javase/7/docs/api/java/lang/String.html> provides complete documentation of String class along with methods.

Let's look into few methods of String

- 1) **concat()**
- 2) **substring()**
- 3) **replace()**
- 4) **contains()**
- 5) **split()**
- 6) **length()**

Concat

```
String s = "Hello";  
  
System.out.println( s.concat(" World"));  
  
// Hello World
```

Substring

```
String s = "ITVedant";  
  
System.out.println(s.substring(0,2));  
  
// IT
```

Replace

```
String s = "I am Batman";  
  
System.out.println( s.replace("Bat", "Super"));
```

//Superman ***Contains***

```
String s = "Superman";  
  
System.out.println( s.contains("man"));  
  
// true
```

Split

```
String s = "I am Batman";  
  
String[] elements = s.split(" ");  
  
for(String element: elements) {  
    System.out.println(element);  
}  
  
// I  
// am  
// Batman
```

Length

```
String s = "ITVedant";  
  
System.out.println(s.length());  
  
// 8
```


Questions

Given -

String s = "I am a java developer";

Guess output of following code

s.substring(7,11);

s.replace("Java", "Android");

s.concat(" and java trainer");

s.concat(" and java trainer");

s.split(" ");

s.charAt(12);

Given

```
String s1 = "Java";
```

```
String s2 = new String("Java");
```

```
String s3 = "Java";
```

```
String s4 = "Python";
```

Guess output of the code

```
s1 == s2;
```

```
s1.equals(s2);
```

```
s1 == s3;
```

```
s1.equals(s3);
```

```
s1 == s4;
```

```
s1.equals(s4);
```

10.5 StringBuffer and StringBuilder

- Unlike String, StringBuffer and StringBuilder are **mutable**.
- The difference between StringBuffer and StringBuilder is that StringBuffer is **thread safe** or **synchronized**.
- StringBuilder is faster than StringBuffer.
- To understand the term **thread safe** or **synchronized** we have to wait for **multi-threading** chapter.
- Both StringBuffer and StringBuilder have lot of inbuilt methods, but most commonly used method is **append()**, which is not available in String.
- **append()** method appends to the current object (**mutable**).

```
StringBuffer buffer = new StringBuffer("Hello");
```

```
buffer.append(" World");
```

```
System.out.println(buffer);
```

```
// Hello World
```

```
StringBuilder builder = new StringBuilder("I am");
```

```
builder.append(" Batman");
```

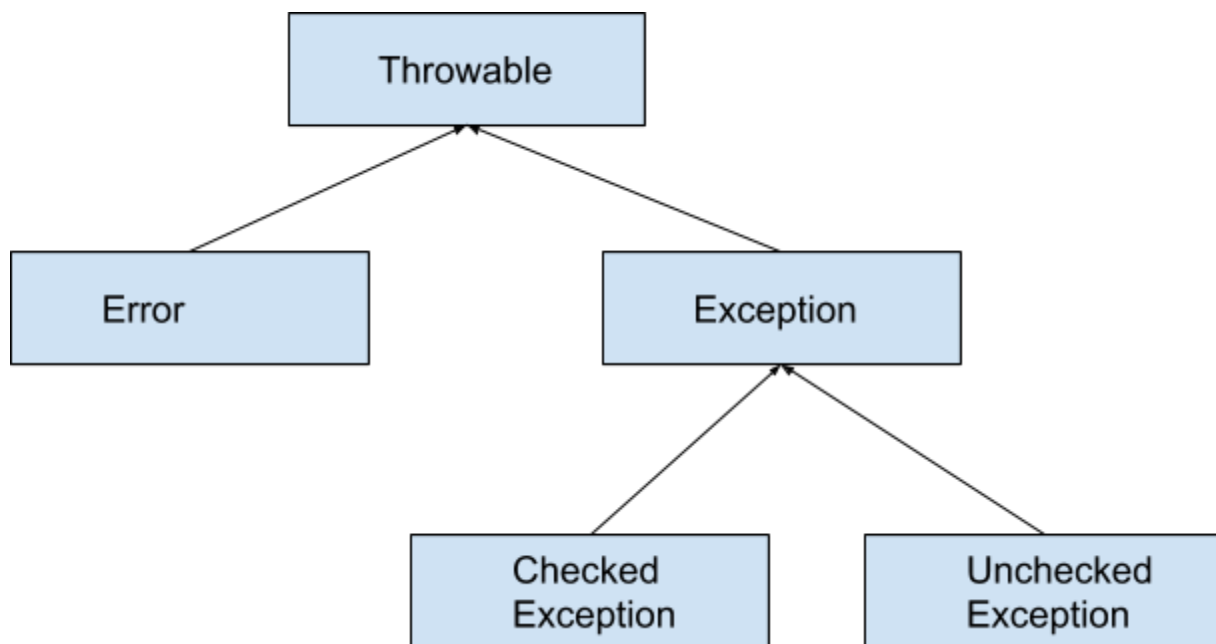
```
System.out.println(builder);
```

```
// I am Batman
```

11. Exception Handling

- An exception is an unwanted situation that disrupts the normal flow of the program.
- Exception occurs at run time.

11.1 Exception hierarchy



Checked exception

- IOException
- FileNotFoundException
- EOFException
- SQLException

Unchecked exception

- ArithmeticException
- NullPointerException
- IndexOutOfBoundsException

- ClassCastException

11.2 Handling Exception (try-catch-finally)

Consider the code

```
System.out.println("Hello world");  
  
int i = 10;  
  
i = i / 0;  
  
System.out.println(i);  
  
System.out.println("Code continues");
```

Output -

Hello world

*Exception in thread "main" java.lang.ArithmeticException: / by zero
at com.itv.test.Test.main(Test.java:11)*

- The above code shows that JVM throws an exception when we try to divide a number by 0 at line 11.
- It throws an object of **ArithmeticException** with message - **/ by zero**.
- Disrupts the normal flow of the program.

Handling exception with try-catch block

```
System.out.println("Hello world");

int i = 10;

try {

    i = i / 0;

    System.out.println(i);

} catch ( ArithmeticException e ) {

    System.out.println("Cannot divide a number by zero");

}

System.out.println("Code continues");
```

Output -

Hello World

Cannot divide a number by zero

Code continues

- A piece of code that is prone to exception should be kept in **try** block.
- The **catch** block catches the exception object and executes only when exception occurs.
- Program continues normally

Handling multiple exceptions with multiple try-catch block

```
System.out.println("Hello world");

int i = 10;

int[ ] array = { 1, 2, 3, 4, 5 };

try {

    System.out.println( array[10] );

    i = i / 0;

    System.out.println(i);

} catch ( ArithmeticException e ) {

    System.out.println("Cannot divide a number by zero");

} catch (ArrayIndexOutOfBoundsException e) {

    System.out.println("Array element does not exists");

}

System.out.println("Code continues");
```

Output -

Hello World

Array element does not exists

Code continues

Understanding program flow

```
System.out.println("Hello world");
```

```
int i = 10;
```

```
i = i / 0;
```

```
System.out.println(i);
```

```
System.out.println("Code continues");
```

Exception

Program flow
disrupted

When surrounded with try-catch block


```
System.out.println("Hello world");
```

```
int i = 10;
```

```
try {
```

```
    i = i / 0;
```

```
    System.out.println(i);
```

```
} catch ( ArithmeticException e ) {
```

```
    System.out.println("Cannot divide  
a number by zero");
```

```
}
```

```
System.out.println("Code continues");
```



Exception
Jumps to next immediate
catch block



Program flows
normally

When surrounded with multiple try-catch block

```
System.out.println("Hello world");
```

```
int i = 10;
```

```
int[] array = { 1, 2, 3, 4, 5 };
```

```
try {
```

```
    System.out.println( array[10] );
```

```
    i = 10 / 0;
```

```
    System.out.println(i);
```

```
} catch ( ArithmeticException e ) {
```

```
    System.out.println("Cannot divide a number by  
zero");
```

```
} catch ( ArrayIndexOutOfBoundsException e ) {
```

```
    System.out.println("Array element does not  
exists");
```

```
}
```

```
System.out.println("Code continues");
```

Exception,
jumps to next
immediate catch
block

Since exception
type is different,
jumps to next
catch block

Finally block

- **Finally block** is a code block that always executes, regardless of exception occurred or not.
- Finally block is used to clear resources, like closing connection or streams.

```
try {  
    System.out.println(1/0);  
} catch (Exception e) {  
    System.out.println( "Cannot divide by zero" );  
} finally {  
    System.out.println( "Hello from finally" );  
}
```

Output -

Cannot divide by zero
Hello from finally

```
try {  
    System.out.println(1/1);  
} catch (Exception e) {  
    System.out.println( "Cannot divide by zero" );  
} finally {  
    System.out.println( "Hello from finally" );  
}
```

Output -

1
Hello from finally

Possible try-catch-finally blocks

```
try {  
}
```



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



```
finally {  
}
```



Correct combinations

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

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

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

11.3 Checked and Unchecked exception

1) Checked exceptions

- Also known as compile time exceptions because these exceptions are checked by the compiler at compile time.
- Compiler gives error if a code is not wrapped under try-catch block.
- **But exception occurs only at run time.**
- Examples-
 - i) `IOException`
 - ii) `FileNotFoundException`
 - iii) `EOFException`
 - iv) `SQLException`

2) Unchecked exceptions

- Also known as run time exceptions, as JVM directly throws these exception at run time.
- Compiler do not check if code is wrapped under try-catch or not
- Examples -
 - i) **ArithmeticException**
 - ii) **NullPointerException**
 - iii) **IndexOutOfBoundsException**
 - iv) **ClassCastException**

We have already seen examples of **unchecked** exceptions like **Arithmetic exceptions** and **ArrayIndexOutOfBoundsException**.

We will see examples of **checked** exceptions in the next chapter.

11.4 Throw and throws keyword

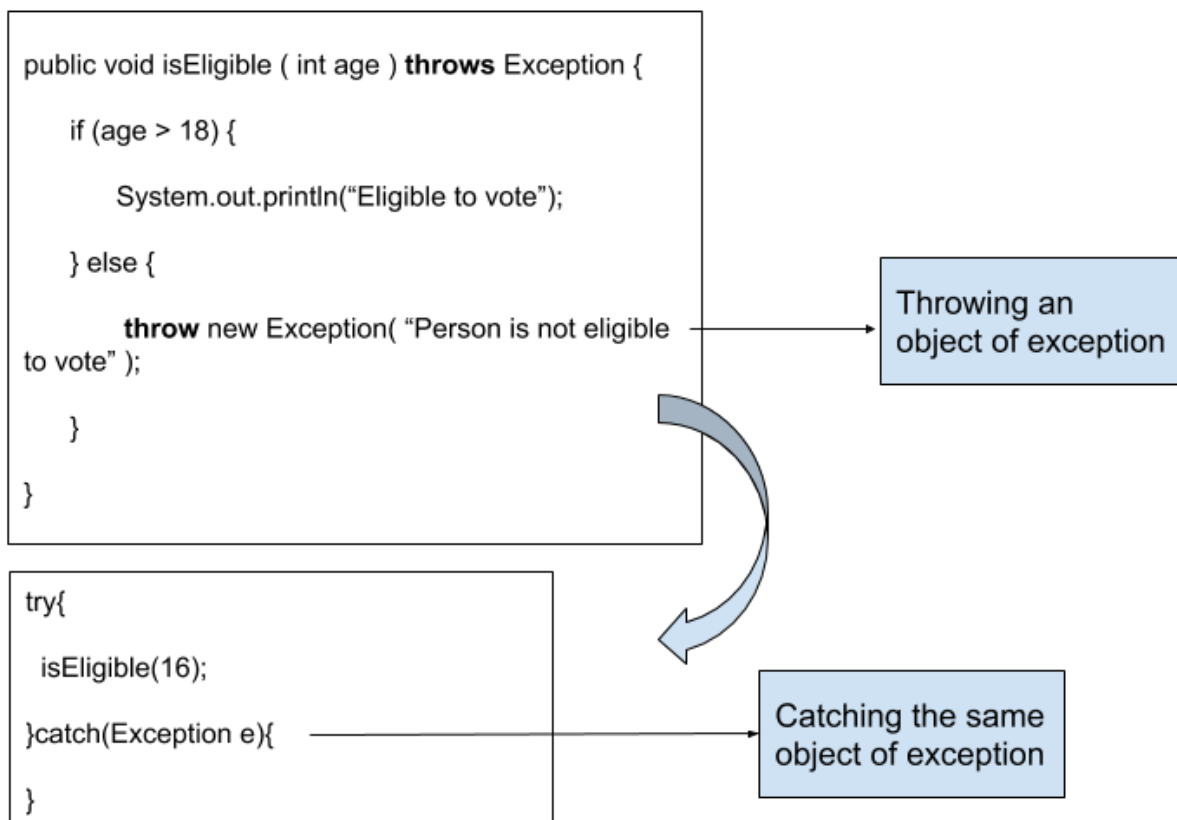
throw keyword is used to intentionally throw an exception.

```
int age = 20;
if (age > 18) {
    System.out.println("Eligible to vote");
} else {
    throw new Exception( "Person is not eligible to vote" );
}
```

In the above example we can see that, we **throw an object of Exception class**. The same object is **caught by the catch block**. This is what JVM does when an exceptional situation occurs.

throws keyword is used as a method signature, to tell the user or developer that this method might throw an exception.

```
public void isEligible ( int age ) throws Exception {  
    if (age > 18) {  
        System.out.println("Eligible to vote");  
    } else {  
        throw new Exception( "Person is not eligible to vote" );  
    }  
}
```



11.5 Creating custom exception

A custom exception can be created by simply extending **Exception** class.

```
class NotEligibleException extends Exception {  
  
    NotEligibleException( String msg ){  
        super( msg );  
    }  
  
}
```

```
public void isEligible ( int age ) throws NotEligibleException {  
  
    if (age > 18) {  
        System.out.println("Eligible to vote");  
    } else {  
        throw new NotEligibleException ( "Person is not eligible to vote" );  
    }  
}
```

Hands on practical assignment

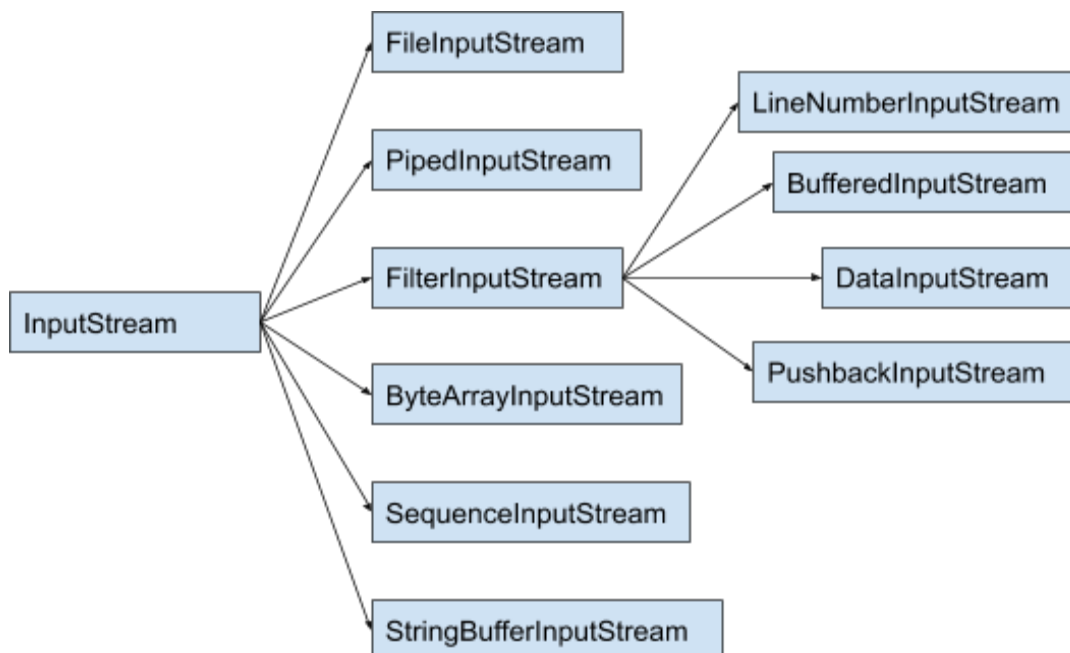
Create a netbanking application. Create Account class with public properties and public getters and setters. Transfer amount between two Account objects. Create and throw custom exceptions for invalid pin and insufficient number.

12. File Handling and I/O

12.1 Streams

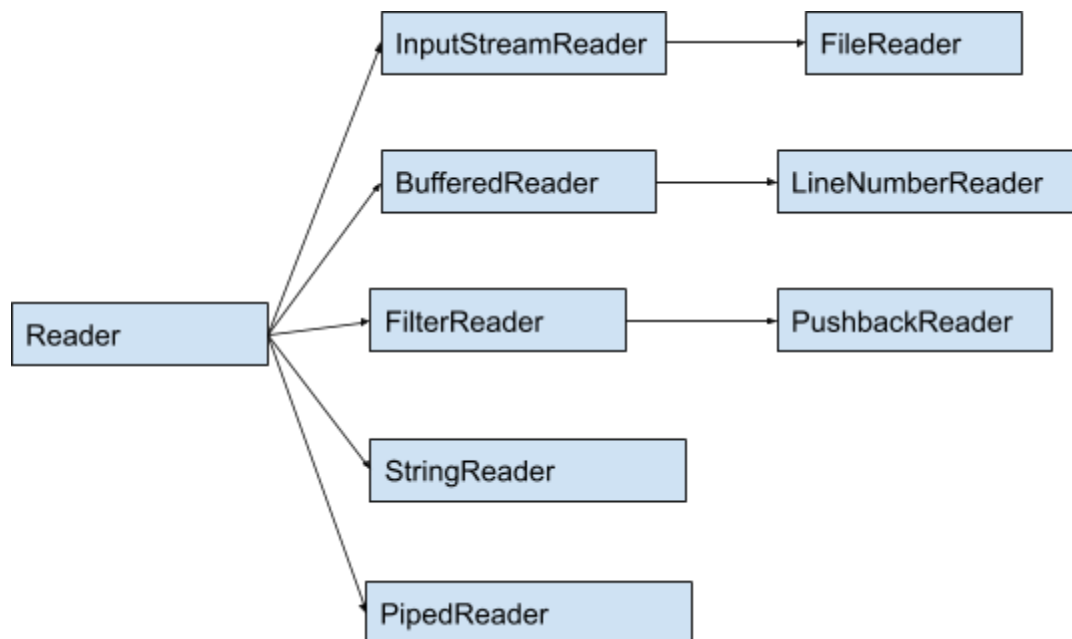
- A sequence of data is known as **stream**.
- We can read from a stream and write to a stream.
- A stream is either connected to a **source** or a **destination** to perform **read** or **write (i/o)** operation.
- **InputStream** is used to read data from source and **OutputStream** is used to write data to destination.
- Both InputStream and OutputStream are **abstract class**.
- InputStream and OutputStream perform on **bytes (8-bit)**.
- **Reader and Writer** abstract class adds the concept to **character (16 bit unicode)** on top of streams.
- They have encoding component that converts character to bytes and bytes to characters.
- All classes related to streams, readers and writers belong to **java.io** package.

Streams hierarchy



OutputStream has same hierarchy.

Character stream hierarchy



Writer has similar hierarchy

Standard streams

Almost all programming language provides standard I/O streams. Java provides three standard streams by default.

- 1) **System.in**
- 2) **System.out**
- 3) **System.err**

System.in is used to feed user's input from keyboard to the program.

System.out is used to display program's output on screen (console).

System.err is used to display error message (in red color) on screen (console).

We already have used **System.out** multiple times, hence let's focus on **System.in** and **System.err**

```
InputStream is = System.in;  
  
InputStreamReader isr = new InputStreamReader(is);  
  
System.out.println("Enter a character");  
  
try {  
    char c = (char) isr.read();  
    System.out.println(c);  
} catch (IOException e) {  
}
```

- The above program creates a stream **System.in** to read data from console.
- **InputStreamReader** is used to convert the byte data to character.
- Similar program can be written to write data to stream **System.out** using **OutputStreamWriter**.
- Let's see an example of **System.err**.

```
OutputStream os = System.err;  
  
OutputStreamWriter osr = new OutputStreamWriter(os);  
  
try {  
    osr.write("This is an error message");  
    osr.flush();  
} catch (IOException e) {  
}
```

- The above program prints messages in **red** color.
- **flush()** method is used write the intended data from buffer memory to the destination.
- Just like **System.out.println()**, above program can also be written as **System.err.println()**.

Scanner and Console Api

Instead of **InputStreamReader**, we can use **Scanner** and **Console** api, which are much simpler to use.

```
Console console = System.console();

if(console != null) {

String username = console.readLine();

char[] password = console.readPassword();

}

Scanner scanner = new Scanner(System.in);

String name = scanner.next();

int age = scanner.nextInt();

double salary = scanner.nextDouble();
```

Console

- Console provides **readLine()** that can read an entire line.
- **readPassword()** can read and store password in char array.
- But Eclipse might not support console api, hence check if it is **null** or not before using.
- Console class also belongs to **java.io** package.

Scanner

- No issue of Scanner api in eclipse.
- It provides **next()** method to read String, **nextInt()** to read int and so on.
- Scanner class do not belong to java.io package, it belongs to **java.util** package.

12.2 FileInputStream and FileOutputStream

- **FileInputStream** and **FileOutputStream** are used perform read and write (i/o) operations with file.
- FileInputStream is used to read data from file.
- FileOutputStream is used to write data to file.

Example code to copy content of one file to another

```
InputStream is = new FileInputStream("fileA.txt");  
OutputStream os = new FileOutputStream("fileB.txt");  
  
int i = 0;  
while ((i = is.read()) != -1) {  
    os.write(i);  
}
```

Hands on practical assignment

1. Integrate Scanner API with previous Netbanking application to read data like amount, pin, account number from user.
2. Write program to copy image, audio, video file.

12.3 File, FileReader and FileWriter

- **File** class provides inbuilt methods to work with files and directories.
 - 1) Creating file.
 - 2) Checks if file exists or not.
 - 3) Checks file permissions to read or write.
 - 4) Set permissions to read or write.
 - 5) Delete file and many more.
- But file class provides no method to read or write character data from file.
- **FileReader** class is used to read data from file.
- **FileWriter** class is used to write data to file.

```
File file = new File("d://myfile.txt");
```

```
if( ! file.exists() ) {
```

```
    try {
```

```
        file.createNewFile();
```

```
    } catch (IOException e) {
```

```
        e.printStackTrace();
```

```
    }
```

```
}
```

Question

Create a file object, write the output after calling the following file methods

Method	Output
canWrite()	
canRead()	
getAbsolutePath()	
delete()	

Reading a file

```
FileReader fr = new FileReader("myfile.txt");  
  
BufferedReader br = new BufferedReader(fr);  
  
String line = " ";  
  
while((line = br.readLine()) != null) {  
    System.out.println(line);  
}
```

Writing to a file

```
FileWriter fw = new FileWriter("myfile.txt");  
  
BufferedWriter bw = new BufferedWriter(fw);  
  
bw.write("This is a new paragraph");  
  
bw.flush();
```

- **BufferedReader** and **BufferedWriter** are used along with **FileReader** and **FileWriter** respectively, to **improve performance**.
- **FileReader** throws **FileNotFoundException** if mentioned file do not exist.
- **FileWriter** creates a **new file** if mentioned file do not exist.
- **FileWriter** by default **rewrites** to the existing file.
- In order to **append** to the existing file, pass an argument **true** along with file name in the **FileWriter** constructor.

```
FileWriter fw = new FileWriter("d://myfile.txt", true);
```

12.4 Serialization and Deserialization

- **Serialization** is a mechanism of converting an object into a byte stream.
- Objects are serialized in order to be stored in a file or sent over a network.
- **Deserialization** is the reverse process of converting a byte stream to an object.

Serialization

```
FileOutputStream fos = new FileOutputStream("objects.ser");  
  
ObjectOutputStream oos = new ObjectOutputStream(fos);  
  
oos.writeObject(new String("Hello"));  
  
oos.flush();
```

- In the above example, Serialization is used to store objects into a file. In this case **ObjectOutputStream** is used along with **FileOutputStream**.
- By convention, file in which objects are stored should be named with **.ser** extension.

Deserialization

```
FileInputStream fis = new FileInputStream("objects.ser");  
  
ObjectInputStream ois = new ObjectInputStream(fis);  
  
String data = (String) ois.readObject();
```

- In the above example, deserialization is used to retrieve objects from a file.
- In this case **ObjectInputStream** is used along with **FileInputStream**.

Serializing a custom object

```
class Employee {  
  
    int id;  
    String name;  
    double salary;  
  
}
```

```
Employee employee = new Employee();  
  
employee.id = 1;  
employee.name = "Batman";  
employee.salary = 25000;  
  
FileOutputStream fos = new FileOutputStream("objects.ser");  
  
ObjectOutputStream oos = new ObjectOutputStream(fos);  
  
oos.writeObject(employee);  
  
oos.flush();
```

The code above throws an exception

```
java.io.NotSerializableException: com.itv.test.Employee  
    at java.base/java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1185)  
    at java.base/java.io.ObjectOutputStream.writeObject(ObjectOutputStream.java:349)  
    at com.itv.test.Test.main(Test.java:25)
```

- The exception says that Employee object **is not serializable**.
- We have to explicitly make Employee object serializable by implementing **Serializable interface**.
- Serializable is a **marker interface or tagging interface** that has no abstract methods, but used to mark or tag an object serializable.

Transient variable

- In case if we don't wish to store any particular property of an object, then that property should be marked transient with a **transient** keyword.
- In the above example, employee's salary can be a private data, let's mark it transient.
- Transient variables return **default value** of their respective data type when deserialized.

```
class Employee implements Serializable {  
  
    int id;  
    String name;  
    transient double salary;  
  
}
```

```
FileInputStream fis = new FileInputStream("objects.ser");  
ObjectInputStream ois = new ObjectInputStream(fis);  
Employee employee = (Employee) ois.readObject();  
System.out.println(employee.id);  
System.out.println(employee.name);  
System.out.println(employee.salary); //returns 0.0
```

Hands on practical assignment

Serialize and store the Account objects from the previous Netbanking application. Retrieve the Account objects from the same file at the beginning of the application.

13. Multi-threading

Multitasking	Multithreading
Multitasking is a OS level feature	Multithreading is a process level feature
Multiple processes(task) can run simultaneously	Multiple threads of a single process can run simultaneously
Separate memory and resources are allocated to each processes	Multiple threads of a single process share the same memory and resources

Thread - A thread is a **light weight process** that has its own call stack.

13.1 Creating threads in Java

Extending thread class

```
class Task extends Thread {  
  
    public void run(){  
  
        //Task to perform  
  
    }  
}
```

```
Task task = new Task();
```

```
task.start();
```

- Thread can be created by **extending thread** class.
- Override **run()** method and write the code that thread need to execute.
- Create object of the class (that extends Thread) and call start method.
- **start()** method executes the code present in **run()** method, but in a separate thread.
- You can also call the **run()** method, but that will execute the code in the same thread.

Implementing runnable interface

```
class Task implements Runnable {  
    public void run(){  
        //Task to perform  
    }  
}
```

```
Task task = new Task();
```

```
Thread thread = new Thread( task );
```

```
thread.start();
```

- Thread can also be created by **implementing Runnable** interface.
- **Runnable** is a **functional interface** with only one abstract method **run()**.
- Override **run()** method and write the code that thread need to execute.
- Create object of the class (that implements Runnable).
- But there is no start method in **Runnable** interface, hence you need to create an object of **Thread** class and pass the object of your class in the **Thread** class constructor.
- Call **start()** method.

Question

According to your observation, which is the best way of creating thread and why ?

13.3 Thread methods

```
class Test {  
    public static void main(String[] args) {  
        Thread thread = Thread.currentThread();  
        System.out.println(thread.getName());  
        System.out.println(thread.getId());  
        System.out.println(thread.getPriority());  
    }  
}
```

//Output

```
main  
1  
5
```

- **currentThread()** is a static method that returns the reference of the currently executing thread.
- In the example above, **main method** is the current thread.
- **getName()** returns the default name of the thread given by JVM.
- **getId()** returns an integer id value given by JVM.
- **getPriority()** method returns value between **1 to 10**. Default priority value is **5**.
- **10** is **MAX_PRIORITY** and **1** is **MIN_PRIORITY**.

Question

Create a 2 thread objects, write the output after calling the following thread methods

Method	Output
getName()	
getId()	
getPriority()	

Important -

- 1) We can set custom name and priority of a thread, but cannot set custom **id**. Hence we have **setName()** and **setPriority()** method, but no **setId()** method.

Write program to set custom name and priority to threads

13.2 Thread life cycle

```
class Task implements Runnable {  
  
    @Override  
    public void run() {  
  
        Thread thread = Thread.currentThread();  
  
        for(int i=0;i<=10;i++) {  
  
            System.out.println(thread.getName() +" "+ i);  
  
        }  
  
    }  
  
}
```

```
class Test {  
  
    public static void main(String[] args) {  
  
        Task task = new Task();  
  
        Thread thread1 = new Thread(task);  
        Thread thread2 = new Thread(task);  
        Thread thread3 = new Thread(task);  
  
        thread1.setName("A");  
        thread2.setName("B");  
        thread3.setName("C");  
  
        thread1.setPriority(Thread.MAX_PRIORITY);  
  
        thread3.setPriority(Thread.MIN_PRIORITY);  
  
    }  
  
}
```

```
        thread1.start();
        thread2.start();
        thread3.start();

    }

}
```

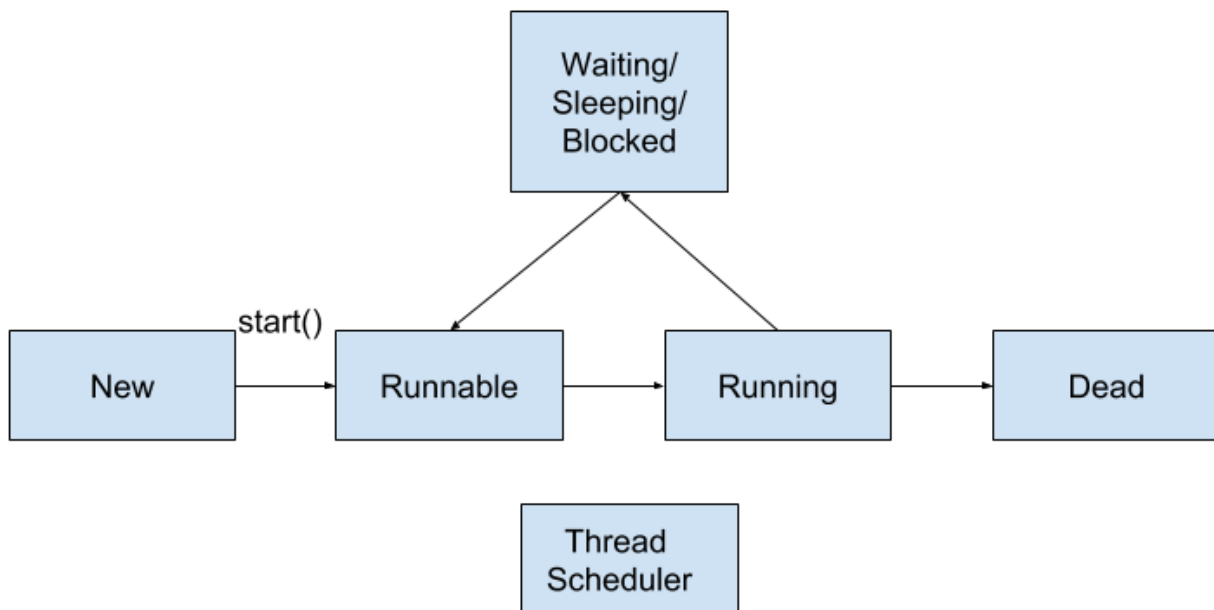
Output

```
B 0
B 1
B 2
B 3
B 4
B 5
B 6
B 7
B 8
B 9
B 10
C 0
A 0
A 1
A 2
A 3
A 4
A 5
C 1
A 6
C 2
A 7
C 3
A 8
A 9
```

A 10
C 4
C 5
C 6
C 7
C 8
C 9
C 10

Conclusions from above code example

- CPU keeps switching between threads.
- **Thread priority do not guarantee** their order of execution.
- CPU can switch to another thread even before current thread completes its execution.
- The order of execution and number of CPU cycle for each thread is controlled by **Thread scheduler**.



New

- When a thread object is created, it is said to be in **new** state.

```
Thread t1 = new Thread();
```

```
Thread t2 = new Thread();
```

- Thread **t1** and **t2** are in new state.

Runnable

- When a thread calls `start()` method, it is said to be in **runnable** state.

```
t1.start();
```

```
t2.start();
```

- Thread **t1** and **t2** are in **runnable** state.

Running

- At this point **thread scheduler** decides to which thread memory will be allocated for execution
- Let's assume **t1** executes first, then **t1** is in current **running** state.

Waiting

- After few CPU cycles, **thread scheduler** decides to allocate memory to **t2** thread for execution.
- Then **t2** is in current running state
- And **t1** moves to **waiting** state.

Dead

- State of a thread keeps moving between **runnable -> running -> waiting -> runnable**.
- Finally when a thread completes its execution, it goes to **dead** state.

Sleeping

- A thread goes to **sleeping** state for some milliseconds when **sleep()** method is called.
- **sleep()** method is a **static** method of **Thread class**.

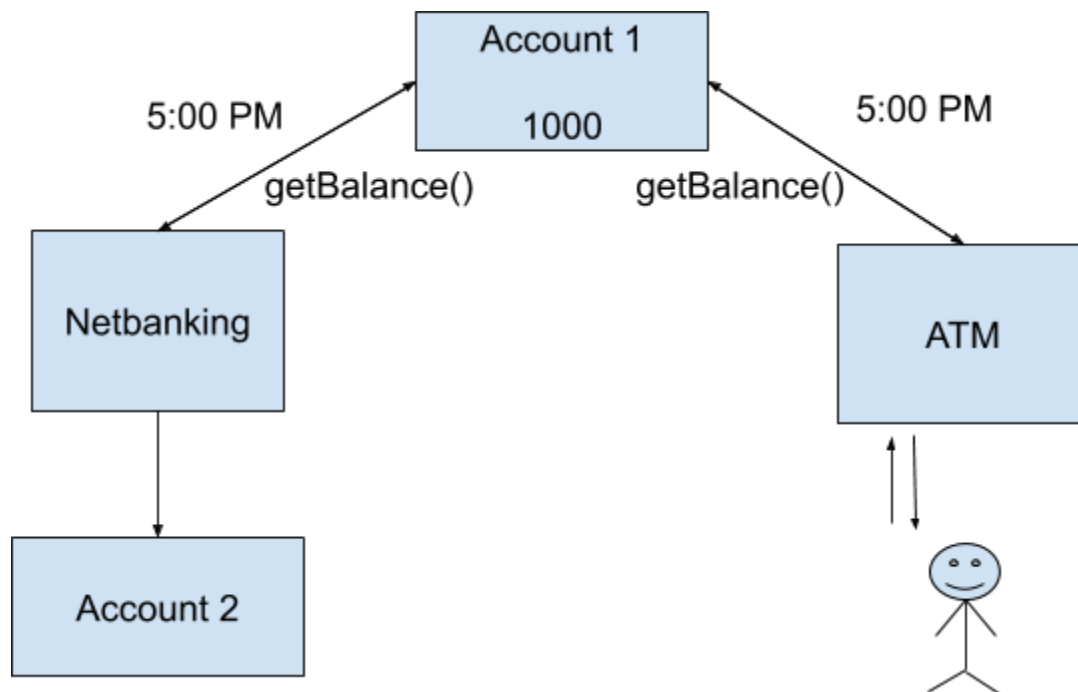
Blocked

We will understand **blocked** state in the next topic - **Synchronization**

Hands on practical assignment

Show a demo example code of **sleep()**, **yield()** and **join()** method.

13.4 Synchronization



- Consider the above situation, given an account with balance **1000**.
- **Person A** tries to transfer that amount to another account through **net banking**.
- At the same time **person B** decides to withdraw the same amount from an **ATM**.
- Here ATM and net banking are two different threads sharing a common resource at same time.
- Let's consider both of them initiate their transaction at sharp **5:00pm** and both receive same amount **1000** each.
- This is not a pleasant situation for the bank, neither for any application.
- This situation can be resolved using **synchronization**.
- Let see a practical example of this situation

Before Synchronization

Account

```
class Account {  
    private double balance;  
  
    public double getBalance() {  
        return balance;  
    }  
  
    public void setBalance( double balance ) {  
        this.balance = balance;  
    }  
}
```

FundTransferTask

```
class FundTransferTask implements Runnable {  
  
    Account sender;  
    Account receiver;  
    double amount;  
  
    public FundTransferTask( Account sender, Account receiver, double amount ) {  
  
        this.sender = sender;  
        this.receiver = receiver;  
        this.amount = amount;  
  
    }  
}
```

```

@Override
public void run() {

    Thread thread = Thread.currentThread();

    double senderBalance = sender.getBalance();

    double receiverBalance = receiver.getBalance();

    if( ( senderBalance - amount ) >= 0 ) {

        receiver.setBalance(receiverBalance + amount);

        sender.setBalance(senderBalance - amount);

    }

    System.out.println(thread.getName()
        +"\n sender balance: "+sender.getBalance()
        +"\n receiver balance: "+receiver.getBalance());

}

}

```

Test

```

class Test {

    public static void main(String[] args) {

        Account sender = new Account();
        sender.setBalance(1000);

        Account receiver1 = new Account();
        receiver1.setBalance(0);

        Account receiver2 = new Account();
        receiver2.setBalance(0);

    }

}

```

```
        Thread t1 = new Thread(new FundTransferTask( sender,receiver1,1000 ));  
        Thread t2 = new Thread(new FundTransferTask( sender,receiver2,1000 ));  
        t1.setName("Netbanking");  
        t2.setName("ATM");  
        t1.start();  
        t2.start();  
    }  
}
```

output

```
Netbanking  
sender balance: 0.0  
receiver balance: 1000.0  
ATM  
sender balance: 0.0  
receiver balance: 1000.0
```

- The output shows that both ATM and netbanking threads receive the same amount.
- This is because they call the `getBalance()` method at the same time and it returns the same amount for both.
- Hence to solve this problem, only one thread should be allowed to access **`getBalance()`** method at a time.
- This is possible by marking `getBalance()` method with **`synchronized`** keyword.

Synchronization is a concept of controlling access of multiple threads to any shared resource

After Synchronization

Account

```
class Account {  
    private double balance;  
  
    public synchronized double getBalance() {  
        return balance;  
    }  
  
    public void setBalance( double balance ) {  
        this.balance = balance;  
    }  
}
```

output

```
ATM  
sender balance: 0.0  
receiver balance: 0.0  
Netbanking  
sender balance: 0.0  
receiver balance: 1000.0
```

- After marking getBalance() method synchronized, we see that only one thread receives the amount.
- In this situation **ATM** thread **acquires the lock** at the beginning and netbanking thread goes into a **blocked** state.
- When ATM thread is done with its execution, **netbanking** thread **acquires the lock**.

Important -

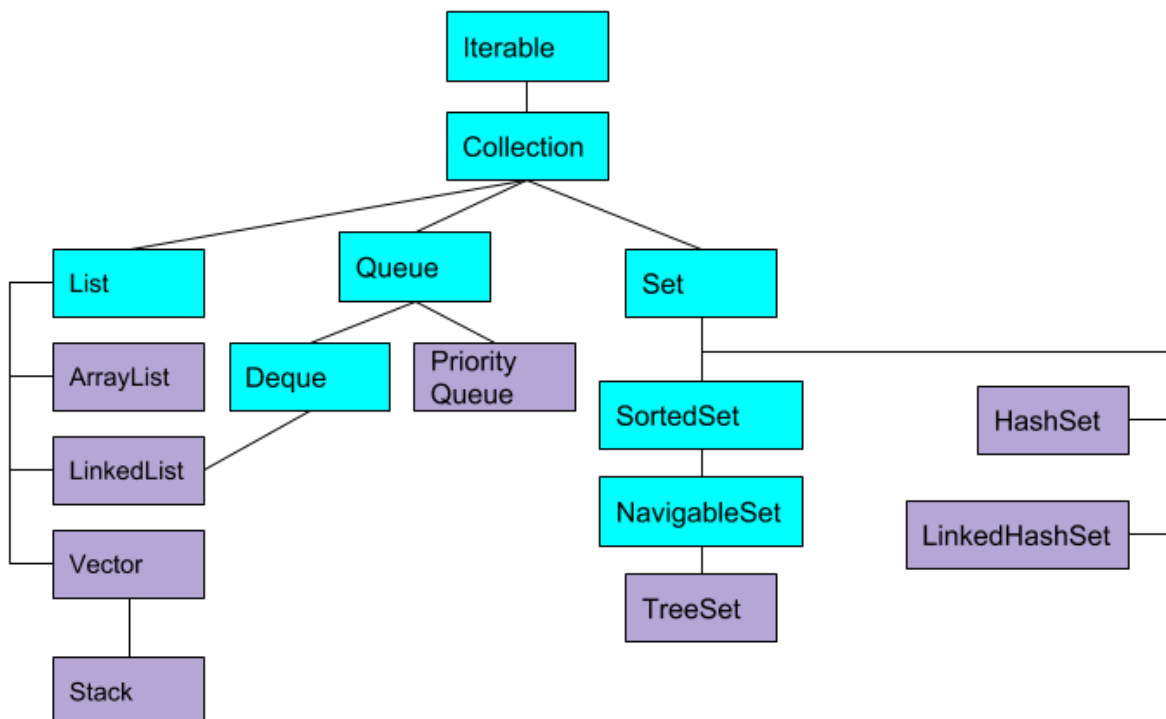
Do you remember, we defined the difference between StringBuffer and StringBuilder. **StringBuffer is thread safe**, that actually means all the **methods in StringBuffer are synchronized**.

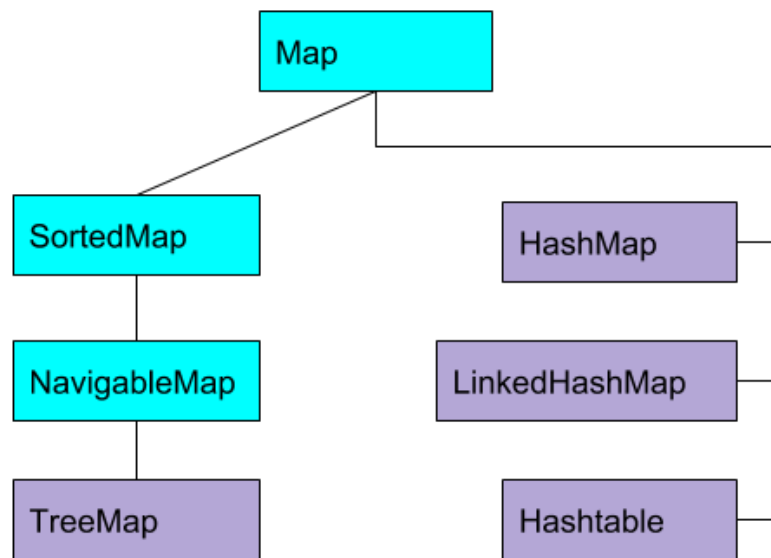
14. Collection Framework

Java Collection framework is a set of interfaces and classes that provides an architecture to store and manipulate group of objects.

Array	Collection Framework
Arrays have fixed size	Data structures in Collection framework are dynamic in size
Arrays can store only homogeneous data	Can store both homogeneous as well as heterogeneous data
Array objects do not have useful inbuilt methods	Data structures in Collection framework have a lot of inbuilt methods to perform various operations

Collection Hierarchy





14.2 List

- **List** is an **ordered** collection, also known as a sequence.
- List allows **positional access** and **search**.
- List can store **duplicate values**.

14.2.1 ArrayList

//Creating list object

```
List myList = new ArrayList();
```

//Adding objects

```
myList.add(1);  
myList.add("ITVedant");  
myList.add(true);  
myList.add(4.5);
```

//Adding objects at specific index

```
myList.add(1,"Batman");
```



```
myList.add(0,"Superman");
```

//Get specific element from list using index

```
System.out.println(myList.get(0));  
System.out.println(myList.get(2));
```

//Remove object by index and object

```
myList.remove(0);  
myList.remove("Superman");
```

//Printing all the objects in the list using for loop
//size method returns size of the list

```
for(int i=0;i<myList.size();i++) {  
    System.out.println(myList.get(i));  
}
```

14.2.2 Generics

- The above code shows that, in collections we can store data of any type (heterogeneous).
- If we wish to restrict this only to a particular data type, this can be achieved using **generics**.

```
List<String> myList = new ArrayList<String>();
```

```
myList.add("Batman");  
myList.add("Superman");  
myList.add(true); //error  
myList.add(4.5); //error
```

14.2.3 LinkedList

//Creating list object

```
List myList = new LinkedList();
```

//Adding objects

```
myList.add(1);  
myList.add("ITVedant");  
myList.add(true);  
myList.add(4.5);
```

//Adding objects at specific index

```
myList.add(1,"Batman");  
myList.add(0,"Superman");
```

//Get specific element from list using index

```
System.out.println(myList.get(0));  
System.out.println(myList.get(2));
```

//Remove object by index and object

```
myList.remove(0);  
myList.remove("Superman");
```

//Printing all the objects in the list using for loop

//size method returns size of the list

```
for(int i=0;i<myList.size();i++) {  
    System.out.println(myList.get(i));  
}
```

The above example shows that **ArrayList** and **LinkedList** have exactly same methods, after all both **implement List** interface. Then what is the difference between them?

ArrayList	LinkedList	Vector
ArrayList internally uses a dynamic resizing array . Internal array resizes by increasing by 50% .	LinkedList internally uses a doubly linklist . A new node is created with every new data inserted.	Vector internally uses a dynamic resizing array . Internal array resizes by increasing by 100% .
Not synchronized	Not synchronized	Synchronized
ArrayList is good for random access, but not good with insertions and deletions.	LinkedList is good for insertion and deletion, but does only sequential access.	Same as ArrayList

Questions

Create a vector object and store String objects (Use generics).

14.2.4 Stack

- **Stack** is a data structure that follows **FILO** or **LIFO** order of operation.
- **FILO** (First In Last Out) or **LIFO** (Last In First Out)
- Stack **extends Vector** and Vector implements List.

```
Stack myStack = new Stack();

//Pushing objects to stack

myStack.push(10);
myStack.push(20);
myStack.push(30);
myStack.push(40);

//Checking object at top

System.out.println(myStack.peek());

//Popping out object at top

System.out.println(myStack.pop());
System.out.println(myStack.pop());
System.out.println(myStack.pop());
System.out.println(myStack.pop());

System.out.println(myStack.pop()); //Empty stack exception
```

Output-

FILO or LIFO order

```
40
30
20
10
Exception
```

14.3 Set

- **Set** is an **unordered** collection.
- Set **do not allow positional access** and **search**.
- Set **cannot store duplicate values**.

14.3.1 HashSet

```
Set<String> mySet = new HashSet<String>();

mySet.add("Superman");
mySet.add("Spiderman");
mySet.add("Ironman");
mySet.add("Batman");
mySet.add("Superman");

mySet.remove("Spiderman");
```

Since Set is an unordered collection, we cannot get index of any element, and hence we cannot use any loops. Now the question is how do we iterate over the set interface?

14.3.2 Iterator

Iterator interface allows one way iteration over any **iterable** object (Hope you remember, **Collection interface extends Iterable interface**)

```
Iterator it = mySet.iterator();

while(it.hasNext()) {

    System.out.println(it.next());

}
```

- **hasNext()** returns boolean, and checks if the collection has an object next to the current object.
- **next()** method returns current object.

Output-

```
Superman  
Batman  
Ironman
```

The above output shows that set removes duplicate values.

Important -

Enhanced for loop (for each loop) internally uses an **iterator**.

14.3.3 LinkedHashSet

```
Set<String> mySet = new LinkedHashSet<String>();  
  
mySet.add("Superman");  
mySet.add("Ironman");  
mySet.add("Batman");  
mySet.add("Superman");  
  
Iterator it = mySet.iterator();  
  
while(it.hasNext()) {  
    System.out.println(it.next());  
}
```

Output

```
Superman  
Ironman  
Batman
```

14.3.4 TreeSet

```
Set<String> mySet = new TreeSet<String>();

mySet.add("Superman");
mySet.add("Ironman");
mySet.add("Batman");
mySet.add("Superman");

Iterator it = mySet.iterator();

while(it.hasNext()) {
    System.out.println(it.next());
}
```

Output-

```
Batman
Ironman
Superman
```

The output of HashSet, LinkedHashSet and TreeSet shows the difference between them.

HashSet	LinkedHashSet	TreeSet
HashSet maintains no order (random) while iterating.	LinkedHashSet maintains insertion order	TreeSet maintains sorted order

Important -

HashSet and **TreeSet** both maintain **sorted order** if data type or object type is **Integer**.

14.3.5 equals() and hashCode() method

Consider the following code

```
class Student {  
  
    private int id;  
    private String name;  
  
    public Student(int id, String name) {  
        super();  
        this.id = id;  
        this.name = name;  
    }  
  
    public int getId() {  
        return id;  
    }  
    public void setId(int id) {  
        this.id = id;  
    }  
  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```

```
Student s1 = new Student(1,"Batman");  
Student s2 = new Student(1,"Batman");  
  
System.out.println( s1.equals(s2) );
```

- Guess the output of the above code.
- Even though both the student object have same value for both the attribute, still **equals()** method returns **false**.
- We already know that **equals()** method belongs to **Object** class.
- Just like **String** class, **Student** class **must override equals()** method to get custom output.

HashCode

- **hashCode()** method returns integer hash code value of an object.
- hashCode() method belongs to **Object** class.
- If two objects are proved equal using **equals()** method, they should also **have same hashCode value**.
- Hence student class must **override hashCode()** method.

```
System.out.println(s1.hashCode());  
System.out.println(s2.hashCode());
```

Output-

```
366712642  
1829164700
```

Consider the following code

```
Set<Student> mySet = new HashSet<Student>();  
  
mySet.add(new Student(1,"Batman"));  
mySet.add(new Student(2,"Superman"));  
mySet.add(new Student(1,"Batman"));  
  
for(Student student: mySet) {  
  
    System.out.println(student.getId());  
    System.out.println(student.getName());  
  
}
```

output-

```
1  
Batman  
1  
Batman  
2  
Superman
```

We know that **Set** interface do not store duplicate objects, still it returns **Student** object with equal attributes twice.

Let's override **equals()** and **hashCode()** method.

```
@Override
public boolean equals(Object obj) {

    Student student = (Student) obj;

    if(student.id == this.id && student.name.equals(this.name)) {

        return true;

    }else {

        return false;

    }

}

@Override
public int hashCode() {

    return this.id*1000;

}
```

Now let's see output of all previous codes

```
Student s1 = new Student(1,"Batman");
Student s2 = new Student(1,"Batman");

System.out.println( s1.equals(s2) );

System.out.println(s1.hashCode());
System.out.println(s2.hashCode());
```

Output-

```
true

1000
1000
```

```
Set<Student> mySet = new HashSet<Student>();

mySet.add(new Student(1,"Batman"));
mySet.add(new Student(2,"Superman"));
mySet.add(new Student(1,"Batman"));

for(Student student: mySet) {

    System.out.println(student.getId());
    System.out.println(student.getName());

}
```

output-

```
1
Batman
2
Superman
```

After overriding **equals()** and **hashCode()** method, Set interface can understand duplicate Student object.

14.3.6 Comparable and Comparator Interface

- **TreeMap** can sort Integer, String, Character and Double objects.
- But it cannot sort any custom object like **Student** object.
- Because TreeMap do not understand how to compare two Student objects.
- Here **Comparable** interface comes in to the picture.
- Student class **need to implement** Comparable interface and **override compareTo()** method.
- **Comparator** is an alternative to Comparable interface.

```
class Student implements Comparable {  
  
    @Override  
    public int compareTo(Object obj) {  
  
        Student student = (Student) obj;  
  
        if(this.id > student.getId()) {  
  
            return 1;  
  
        }else if(this.id < student.getId()) {  
  
            return -1;  
  
        }else {  
  
            return 0;  
  
        }  
    }  
}
```

```
Set<Student> mySet = new TreeSet<Student>();  
  
mySet.add(new Student(2,"Superman"));  
mySet.add(new Student(3,"IronMan"));  
mySet.add(new Student(1,"Batman"));  
  
for(Student student: mySet) {  
  
    System.out.println(student.getId());  
    System.out.println(student.getName());  
  
}
```

output-

```
1
Batman
2
Superman
3
IronMan
```

14.4 Map

- **Map** is an **unordered** collection, that stores data in **key, value** pair.
- In a Map interface, objects can be searched with their corresponding key value.
- Map can have only one value for one key. (No duplicate keys)

14.4.1 HashMap

```
Map<String,String> myMap = new HashMap<String,String>();

myMap.put("India","Kolkata");
myMap.put("USA","Washington DC");
myMap.put("England","London");
myMap.put("India","Delhi");
myMap.put(null, "Mumbai");

System.out.println(myMap.get("India")); // returns Delhi, not Kolkata
System.out.println(myMap.get(null));

myMap.remove("England");
```

In map we can either iterate over **keyset** or **values**.

```
Iterator keyIterator = myMap.keySet().iterator();

while(keyIterator.hasNext()) {

    System.out.println(keyIterator.next());

}
```

```

Iterator valueIt = myMap.values().iterator();

while(valueIt.hasNext()) {

    System.out.println(valueIt.next());

}

```

HashMap	LinkedHashMap	TreeMap	Hashtable
Maintains no order (random) while iterating.	Maintains insertion order	Maintains sorted order	Maintains no order (random) while iterating.
Not Synchronized	Not Synchronized	Not Synchronized	Synchronized
Allow null key	Allow null key	Do not allow null key	Do not allow null key

14.5 Queue

- **Queue** is a data structure that follows **FIFO** order of operation.
- **FIFO** (First In First Out)

14.5.1 LinkedList

LinkedList implements both **List** and **Deque**, **Deque** implements **Queue**.

```

Queue<Integer> queue = new LinkedList<Integer>();

queue.offer(10);
queue.offer(20);
queue.offer(30);
queue.offer(40);

System.out.println(queue.peek());

System.out.println(queue.poll());

```

```
System.out.println(queue.poll());
System.out.println(queue.poll());
System.out.println(queue.poll());
System.out.println(queue.poll()); //returns null
```

FIFO order

Output-

```
10
20
30
40
null
```

14.5.2 PriorityQueue

Priority queue have similar methods but returns output based on **priority**. In this case ordered output.

```
Queue<Integer> queue = new PriorityQueue<Integer>();

queue.offer(20);
queue.offer(30);
queue.offer(10);

System.out.println(queue.peek());

System.out.println(queue.poll());
System.out.println(queue.poll());
System.out.println(queue.poll());
System.out.println(queue.poll()); //returns null
```

Output-

```
10
20
30
null
```