CORE JAVA CHEATSHEET

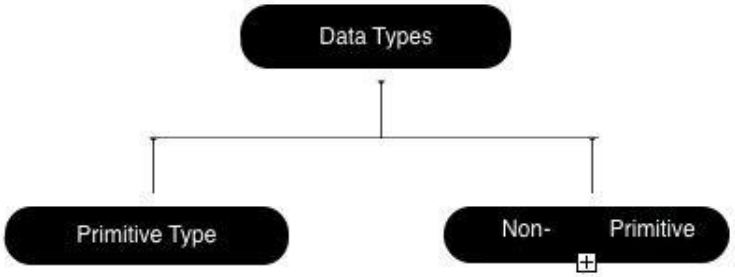
Object Oriented Programming Language: based on the concepts of “objects”.
Open Source: Readily available for development.
Platform-neutral: Java code is independent of any particular hardware or software. This is because Java code is compiled by the compiler and converted into byte code. Byte code is platform-independent and can run on multiple systems. The only requirement is Java needs a runtime environment i.e, JRE, which is a set of tools used for developing Java applications.
Memory Management: Garbage collected language, i.e. deallocation of memory.
Exception Handling: Catches a series of errors or abnormality, thus eliminating any risk of crashing the system.

THE JAVA BUZZWORDS

Java was modeled in its final form keeping in consideration with the primary objective of having the following features:

Simple, Small and Familiar
Object-Oriented
Portable and Platform Independent
Compiled and Interpreted
Scalability and Performance
Robust and Secure
Architectural-neutral
High Performance
Multi-Threaded
Distributed
Dynamic and Extensible

DATA TYPES IN JAVA



Primitive Data Types

Data Type	Default Value	Size (in bytes) 1 byte = 8 bits
boolean	FALSE	1 bit
char	" " (space)	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0	8 byte
float	0.0f	4 byte
double	0.0d	8 byte

Non-Primitive Data Types

Data Type
String
Array
Class
Interface

TYPECASTING

It is a method of converting a variable of one data type to another data type so that functions can process these variables correctly.
Java defines two types of typecasting:

- Implicit Type Casting (Widening)

Storing a variable of a smaller data type to a larger data type.

- Explicit Typecasting (Narrowing)
  Storing variable of a larger data type to a smaller data type.

OPERATORS IN JAVA

Java supports a rich set of operators that can be classified into categories as below :

Operator Category	Operators
Arithmetic operators	+,-,/,*,%
Relational operators	<, >, <=, >=,==, !=
Logical operators	&& , ||
Assignment operator	=, +=, −=, ×=, ÷=, %=, &=, ^=, |=, <<=, >>=, >>>=
Increment and Decrement operator	++ , - -
Conditional operators	?:
Bitwise operators	^, &, |
Special operators	. (dot operator to access methods of class)

JAVA IDE and EXECUTING CODE

Amongst many IDEs the most recommended ones are :

- Eclipse
- NetBeans

Java code can also be written in any text editor and compiled on the terminal with following commands :

```
$ java [file_name].java
```

```
$ java [file_name]
```

Note: File name should be the same as the class name containing the main() method, with a .java extension.

VARIABLES IN JAVA

Variables are the name of the memory location. It is a container that holds the value while the java program is executed. Variables are of three types in Java :

Local Variable	Global or Instance Variable	Static Variable
Declared and initialized inside the body of the method, block or constructor.	Declared inside the class but outside of the method, block or constructor. If not initialized, the default value is 0.	Declared using a “static” keyword. It cannot be local.
It has an access only within the method in which it is declared and is destroyed later from the block or when the function call is returned.	Variables are created when an instance of the class is created and destroyed when it is destroyed.	Variables created creates a single copy in the memory which is shared among all objects at a class level.

```
class TestVariables
{
        int data = 20;                  // instance variable
        static int number = 10;         //static variable

        void someMethod()
        {
                int num = 30;           //local variable
        }
}
```

RESERVED WORDS

Also known as keywords, are particular words which are predefined in Java and cannot be used as variable or object name. Some of the important keywords are :

Keywords	Usage
abstract	used to declare an abstract class.
catch	used to catch exceptions generated by try statements.
class	used to declare a class.
enum	defines a set of constants
extends	indicates that class is inherited
final	indicates the value cannot be changed
finally	used to execute code after the try-catch structure.
implements	used to implement an interface.
new	used to create new objects.
static	used to indicate that a variable or a method is a class method.
super	used to refer to the parent class.
this	used to refer to the current object in a method or constructor.
throw	used to explicitly throw an exception.
throws	used to declare an exception.
try	block of code to handle exception

METHODS IN JAVA

The general form of method :

```
type name (parameter list)
{
        //body of the method

        //return value          (only if type is not void)
}
```

Where type - return type of the method
name - name of the method
parameter list - sequence of type and variables separated by a comma
return - statement to return value to calling routine

CONDITIONAL STATEMENTS IN JAVA

1. **if-else**

Tests condition, if condition true if block is executed else the else block is executed.

```
class TestIfElse
{
        public static void main(String args[])
        {
                int percent = 75;

                if(percent >= 75)
                {
                        System.out.println("Passed");
                }
                else
                {
                        System.out.println("Please attempt again!");
                }
        }

}
```

2. Switch

Test the condition, if a particular case is true the control is passed to that block and executed. The rest of the cases are not considered further and the program breaks out of the loop.

```
class TestSwitch
{
        public static void main(String args[])
        {
        int weather = 0;

        switch(weather)
                {
                case 0 :
                        System.out.println("Sunny");
```

```
                    break;
                case 1 :
                    System.out.println("Rainy");
                    break;
                case 2 :
                    System.out.println("Cold");
                    break;
                case 3 :
                    System.out.println("Windy");
                    break;
                default :
                System.out.println("Pleasant");
                }
        }
}
```

LOOPS IN JAVA

Loops are used to iterate the code a specific number of times until the specified condition is true. There are three kinds of loop in Java :

For Loop	
Iterates the code for a specific number of times until the condition is true.	*class TestForLoop* *{* *public static void main (String args[])* *{* *for(int i=0;i<=5;i++)* *System.out.println("*");* *}* *}*
While Loop	
If condition in the while is true the program enters the loop for iteration.	*class TestWhileLoop* *{* *public static void main (String args[])* *{* *int i = 1;* *while(i<=10)* *{* *System.out.println(i);* *i++;* *}* *}* *}*
Do While Loop	

The program enters the loop for iteration at least once irrespective of the while condition being true. For further iterations, it is depends on the while condition to be true.	*class TestDoWhileLoop* *{* *public static void main (String args[])* *{* *int i = 1;* *do* *{* *System.out.println(i);* *i++;* *}while(i<=10);* *}* *}*

JAVA OOPS CONCEPTS

An object-oriented paradigm offers the following concepts to simplify software development and maintenance.

1. Object and Class

Objects are basic runtime entities in an object-oriented system, which contain data and code to manipulate data. This entire set of data and code can be made into user-defined data type using the concept of class. Hence, a class is a collection of objects of a similar data type.
Example: apple, mango, and orange are members of class fruit.

2. Data Abstraction and Encapsulation

The wrapping or enclosing up of data and methods into a single unit is known as encapsulation. Take medicinal capsule as an example, we don't know what chemical it contains, we are only concerned with its effect.

This insulation of data from direct access by the program is called data hiding. For instance, while using apps people are concerned about its functionality and not the code behind it.

3.Inheritance

Inheritance provides the concept of reusability, it is the process by which objects of one class (Child class or Subclass) inherit or derive properties of objects of another class (Parent class).

Types of Inheritance in Java:

Single Inheritance

The child class inherits properties and behavior from a single parent class.
Multilevel Inheritance
The child class inherits properties from its parent class, which in turn is a child class to another parent class
Multiple Inheritance
When a child class has two parent classes. In Java, this concept is achieved by using interfaces.
Hierarchical Inheritance
When a parent class has two child classes inheriting its properties.

```
class A
{
        int i, j;
        void showij() {
        System.out.println("i and j: " + i + " " + j);
        }
}
// Create a subclass by extending class A.
class B extends A {
        int k;
        void showk()
        { System.out.println("k: " + k);
        }
        void sum()
        { System.out.println("i+j+k: " +
        (i+j+k));
        }
}

class SimpleInheritance {
        public static void main(String args[])
        {A objA = new A();
        B objB = new B();
        // The superclass may be used by itself.
        objA.i = 10;
        objA.j = 20;
        System.out.println("Contents of objA: ");
        objA.showij();
        System.out.println();
        /* The subclass can access to all public members of
        its superclass. */
        objB.i = 7;
        objB.j = 8;
```

```
        System.out.println("Contents of objB: ");
        objB.showij();
        objB.showk();
        System.out.println();
        System.out.println("Sum of i, j and k in objB:");
        objB.sum();
        }
}
```

Some limitations in Inheritance :

- Private members of the superclass cannot be derived by the subclass.
- Constructors cannot be inherited by the subclass.
- There can be one superclass to a subclass.

4. Polymorphism

Defined as the ability to take more than one form. Polymorphism allows creating clean and readable code.
In Java Polymorphism is achieved by the concept of method overloading and method overriding, which is the dynamic approach.

- Method Overriding

In a class hierarchy, when a method in a child class has the same name and type signature as a method in its parent class, then the method in the child class is said to override the method in the parent class.

In the code below, if we don't override the method the output would be 4 as calculated in ParentMath class, otherwise, it would be 16.

```
class ParentMath
{
        void area()
        {
                int a =2;
                System.out.printf("Area of Square with side 2 = %d %n", a * a);
                System.out.println();
        }
}

class ChildMath extends ParentMath
{
```

```
        void area()
        {
                int a =4;
                System.out.printf("Area of Square with side 4= %d %n", a * a);
        }

        public static void main (String args[])
        {
                ChildMath obj = new ChildMath();
                obj.area();
        }
}
```

- Method Overloading

Java programming can have two or more methods in the same class sharing the same name, as long as their arguments declarations are different. Such methods are referred to as overloaded, and the process is called method overloading.

Three ways to overload a method :

1. Number of parameters
   example: add(int, int)
   add(int, int, int)

2. Data type of parameters
   example add(int, int)
   add(int, float)

3.Sequence of data type of parameters
   example add(int, float)
   add(float, int)

Program to explain multilevel inheritance and method overloading :

```
class Shape
{
   void area()
   {
      System.out.println("Area of the following shapes are : ");
   }
}
```

```
class Square extends Shape
{

    void area(int length)
    {
        //calculate area of square
        int area = length * length;
        System.out.println("Area of square : "+area);
    }
}

class Rectangle extends Shape
{
     //define a breadth

    void area(int length,int breadth)
    {
         //calculate area of rectangle
        int area = length * breadth;
        System.out.println("Area of rectangle : " + area);
    }
}

class Circle extends Shape
{
    void area(int breadth)
    {
        //calculate area of circle using length of the shape class as radius
        float area = 3.14f * breadth * breadth;
        System.out.println("Area of circle : " + area);
    }
}

class InheritanceOverload
{
    public static void main(String[] args)
    {
         int length = 5;
         int breadth = 7;
         Shape s = new Shape();
        //object of child class square
        Square sq = new Square();
        //object of child class rectangle
```

```
        Rectangle rec = new Rectangle();
        //object of child class circle
        Circle cir = new Circle();

        //calling the area methods of all child classes to get the area of different objects
        s.area();
        sq.area(length);
        rec.area(length,breadth);
        cir.area(length);
    }
}
```

ABSTRACT CLASS

Superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to implement its methods.

```
abstract class A {
        abstract void callme();
        // concrete methods are still allowed in abstract classes
        void callmetoo() {
        System.out.println("This is a concrete method.");
        }
        }
        class B extends A
        {void callme() {
        System.out.println("B's implementation of callme.");
        }
}

class Abstract {
        public static void main(String args[])
        {B b = new B();
        b.callme();
        b.callmetoo();
        }
}
```

INTERFACES

A class's interface can be full abstracted from its implementation using the "interface" keyword. They are similar to class except that they lack instance variables and their methods are declared without any body.

- Several classes can implement an interface.
- Interfaces are used to implement multiple inheritances.
- Variables are public, final and static.
- To implement an interface, a class must create a complete set of methods as defined by an interface.
- Classes implementing interfaces can define methods of their own.

```
interface Area
{
        final static float pi = 3.14F;
        float compute(float x , float y);
}

class Rectangle implements Area
{
        public float compute (float x, float y)
        {
                return (x*y);
        }
}

class Circle implements Area
{
        public float compute (float x, float y)
        {
                return (pi * x * x);
        }
}

class InterfaceTest
{
        public static void main (String args[])
        {
                float x = 2.0F;
                float y = 6.0F;
                Rectangle rect = new Rectangle();               //creating object
                Circle cir = new Circle();

                float result1 = rect.compute(x,y);
```

```
                System.out.println("Area of Rectangle = "+ result1);

                float result2 = cir.compute(x,y);
                System.out.println("Area of Circle = "+ result2);
        }
}
```

CONSTRUCTORS IN JAVA

- A constructor initializes an object on creation.
- They have the same name as the class.
- They do not have any return type, not even void.
- Constructor cannot be static, abstract or final.
- Constructors can be :
- Non - Parameterized or Default Constructor: Invoked automatically even if not declared

```
class Box {
        double width;
        double height;
        double depth;
        // This is the constructor for Box.
        Box() {
        System.out.println("Constructing Box");
        width = 10;
        height = 10;
        depth = 10;
        }
        // compute and return volume
        double volume() {
        return width * height * depth;
        }
}

class BoxVol {
        public static void main(String args[]) {
        // declare, allocate, and initialize Box objects
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;

        vol = mybox1.volume();
        System.out.println("Volume is " + vol);
```

```
        vol = mybox2.volume();
        System.out.println("Volume is " + vol);
        }
}
```

Parameterized :

Used to initialize the fields of the class with predefined values from the user.

```
class Box
{ double
width; double
height;double
depth;
        Box(double w, double h, double d)
        {width = w;
        height = h;
        depth = d;
        }

        double volume() {
        return width * height * depth;
        }
}
class BoxVolP {
        public static void main(String args[]) {

        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box(3, 6, 9);
        double vol;

        vol = mybox1.volume();
        System.out.println("Volume is " + vol);

        vol = mybox2.volume();
        System.out.println("Volume is " + vol);
        }
```

ARRAYS IN JAVA

Array is a group of like-type variables that are referred by a common name, having continuous memory. Primitive values or objects can be stored in an array. It provides code optimization since we can sort data efficiently and also access it randomly. The only flaw is that we can have a fixed-size elements in an array.

There are two kinds of arrays defined in Java:

1. Single Dimensional: Elements are stored in a single row

```
import java.util.Scanner;

class SingleArray
{
	public static void main(String args[])
	{
		int len = 0;
		//declaration
		int [] numbers = {3,44,12,53,6,87};
		Scanner s = new Scanner(System.in);

		System.out.println("The elements in the array are: ");
		for(int i=0;i<numbers.length;i++)
		{
			System.out.print(numbers[i] + " ");
		}
		System.out.println();

		System.out.println("The sum of elements in the array are: ");
		int sum =0;
		for(int i=0;i<numbers.length;i++)
		{
			sum = sum + numbers[i];
		}
		System.out.println("Sum of elements = " + sum);
	}
}
```

2. Multi-Dimensional: Elements are stored as row and column

```
class MatrixArray
```

```
{
        public static void main(String args[])
        {
                int [][] m1 = { {1,5,7}, {2,4,6}};
                int [][] m2 = {{1,2,1},{4,4,3}};
                int [][] sum = new int [3][3];

                //printing matrix

                System.out.println("The given matrix is : ");
                 for(int a=0;a<=m1.length;a++)
                 {
                        for(int b=0;b<=m2.length;b++)
                        {
                                System.out.print(m1[a][b] + " ");
                        }

                        System.out.println();
                }

                 //matrix addition
                 System.out.println("The sum of given 2 matrices is : ");
```

```
                for(int a=0;a<=m1.length;a++)
                {
                        for(int b=0;b<=m2.length;b++)
                        {
                                sum[a][b] = m1[a][b] + m2[a][b];
                                System.out.print(sum[a][b] + " ");
                        }
                        System.out.println();
                }
        }
}
```

STRINGS IN JAVA

- Strings are non-primitive data type that represents a sequence of characters.
- String type is used to declare string variables.
- Array of strings can also be declared.
- Java strings are immutable, we cannot change them.
- Whenever a string variable is created, a new instance is created.

Creating String

Using Literal	Using new keyword
String name = “John” ;	String s = new String();

String Methods

The String class which implements CharSequence interface defines a number of methods for string manipulation tasks. List of most commonly used string methods are mentioned below:

Method	Task Performed
toLowerCase()	converts the string to lower case
toUpperCase()	converts the string to upper case
replace('x' , 'y')	replaces all appearances of 'x' with 'y'
trim()	removes the whitespaces at the beginning and at the end
equals()	returns 'true' if strings are equal
equalsIgnoreCase()	returns 'true' if strings are equal, irrespective of case of characters
length()	returns the length of string
CharAt(n)	gives the nth character of string
compareTo()	returns negative if string 1 < string 2 positive if string 1 > string 2 zero if string 1 = string 2
concat()	concatenates two strings
substring(n)	returns substring returning from character n
substring(n,m)	returns a substring between n and ma character.
toString()	creates the string representation of object
indexOf('x')	returns the position of first occurrence of x in the string.
indexOf('x',n)	returns the position of after nth position in string
ValueOf (Variable)	converts the parameter value to string representation

Program to show Sorting of Strings:

```
class SortStrings {
        static String arr[] = {
        "Now", "the", "is", "time", "for", "all", "good", "men",
        "to", "come", "to", "the", "aid", "of", "their", "county"
        };
        public static void main(String args[])
        {
                for(int j = 0; j < arr.length; j++)
                {
                for(int i = j + 1; i < arr.length; i++)
                {
                        if(arr[i].compareTo(arr[j]) < 0)
                        {
                                String t = arr[j];
                                arr[j] = arr[i];
```

```
                    arr[i] = t;
                }
            }
            System.out.println(arr[j]);
        }   }
}
```

String Buffer and String Builder

- For mutable strings, we can use StringBuilder and StringBuffer classes which as well implement CharSequence interface.
- These classes represent growable and writable character interface.
- They automatically grow to make room for additions , and often has more characters preallocated than are actually needed, to allow room for growth.

Difference between length() and capacity()

length() : To find the length of StringBuffer
capacity() : To find the total allocated capacity

```
    /* StringBuffer length vs. capacity */
class StringBufferTest {
    public static void main(String args[])
    { StringBuffer sb = new StringBuffer("Hello");
    System.out.println("buffer = " + sb);
    System.out.println("length = " + sb.length());
    System.out.println("capacity = " + sb.capacity());
    }
}
```

StringBuilder versus StringBuffer

String Builder	String Buffer
Non-Synchronized : hence efficient.	Synchronized
Threads are used, multithreading.	Thread Safe

MULTITHREADING

Multitasking: Process of executing multiple tasks simultaneously, to utilize the CPU.

This can be achieved in two-ways:

- Process-based multitasking.(Multitasking)
- Thread-based multitasking (Multithreading)

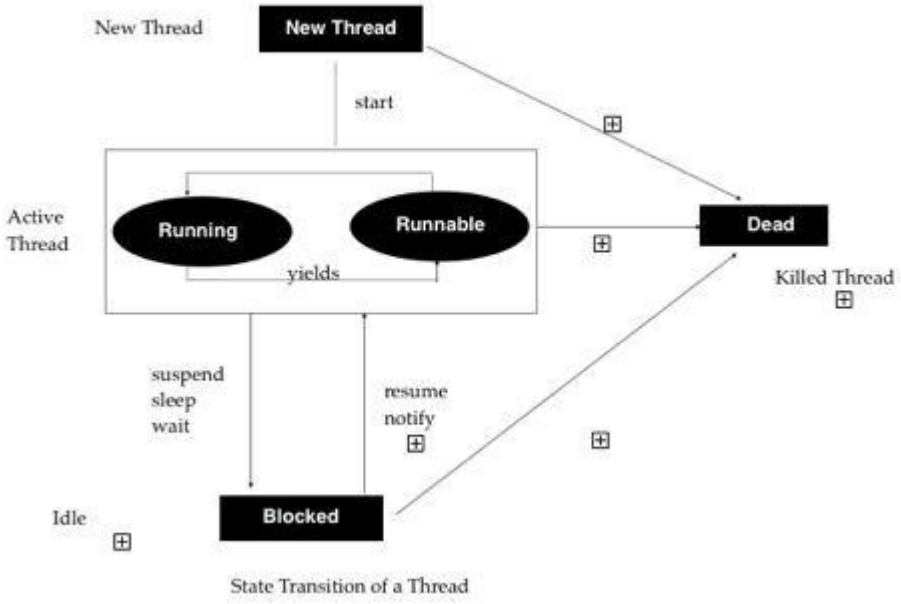
Multitasking versus Multithreading

Multitasking	Multithreading
OS concept in which multiple tasks are performed simultaneously.	Concept of dividing a process into two or more subprocess or threads that are executed at the same time in parallel.
Multiple programs can be executed simultaneously.	Supports the execution of multiple parts of a single program simultaneously.
Process has to switch between different programs or processes.	Processor needs to switch between different parts or threads of the program.
less efficient	highly efficient
program or process in the smallest unit in the environment	thread is the smallest unit
cost effective	expensive

Life Cycle Of Thread

A thread is always in one of the following five states, it can move from state to another by a variety of ways as shown.

New thread: Thread object is created. Either it can be scheduled for running using start() method.
Runnable thread : Thread is ready for execution and waiting for processor.
Running thread: It has got the processor for execution.
Blocked thread: Thread is prevented from entering into runnable state.
Dead state: Running thread ends its life when it has completed executing its run() method.



State Transition of a Thread

Creating Thread

Extending Thread class
Implementing Runnable interface

Common Methods Of Thread Class

Method	Task Performed
public void run()	Inherited by class MyThread It is called when thread is started, thus all the action takes place in run()
public void start()	Causes the thread to move to runnable state.
public void sleep(long milliseconds)	Blocks or suspends a thread temporarily for entering into runnable and subsequently in running state for specified milliseconds.
public void yield	Temporarily pauses currently executing thread object and allows other threads to be executed.
public void suspend()	to suspend the thread, used with resume() method.
public void resume()	to resume the suspended thread
public void stop()	to cause premature death of thread, thus moving it to dead state.

Program to create threads using thread class.

```
class A extends Thread
{
	public void run()

	{
		for(int i=1;i<=5;i++)
		{
			System.out.println("From thread A : i " + i);
		}
		System.out.println("Exit from A ");
	}
}

class B extends Thread
{
	public void run()

	{
		for(int i=0;i<=5;i++)
		{
			System.out.println("From thread B : i " + i);
		}
		System.out.println("Exit from B ");
	}
}

class C extends Thread
{
	public void run ()
	{
		for(int k=1;k<=5;k++)
		{
			System.out.println("From thread C : k " + k);
		}
		System.out.println("Exit from C ");
	}
}

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

```
                new A().start();
                new B().start();
                new C().start();
        }
}
```

Implementing Runnable Interface

The run( ) method that is declared in the Runnable interface which is required for implementing threads in our programs.

Process consists of following steps :

- Class declaration implementing the Runnable interface
- Implementing the run() method
- Creating a thread by defining an object that is instantiated from this “runnable” class as the target of the thread.
- Calling the thread’s start() method to run the thread.

Using Runnable Interface

```
class X implements Runnable
{
        public void run()
        {
                for(int i=0;i<=10;i++)
                {
                        System.out.println("Thread X " + i);
                }
                System.out.println("End of thread X ");
        }
}

class RunnableTest
{
        public static void main(String args[])
        {
                X runnable = new X ();
                Thread threadX = new Thread(runnable);
                threadX.start();
                System.out.println("End of main Thread");
        }
}
```

Thread Class Versus Runnable Interface

Thread Class	Runnable Interface
Derived class extending Thread class itself is a thread object and hence, gains full control over the thread life cycle.	Runnable Interface simply defines the unit of work that will be executed in a thread, so it doesn't provide any control over thread life cycle.
The derived class cannot extend other base classes	Allows to extend base classes if necessary
Used when program needs control over thread life cycle	Used when program needs flexibility of extending classes.

EXCEPTION HANDLING

Exception is an abnormality or error condition that is caused by a run-time error in the program, if this exception object thrown by error condition is not caught and handled properly, the interpreter will display an error message. If we want to avoid this and want the program to continue then we should try to catch the exceptions.This task is known as exception handling.

Common Java Exceptions

Exception Type	Cause of Exception
ArithmeticException	caused by math errors
ArrayIndexOutOfBoundException	caused by bad array indexes
ArrayStoreException	caused when a program tries to store wrong data type in an array
FileNotFoundException	caused by attempt to access a nonexistent file
IOException	caused by general I/O failures.
NullPointerException	caused by referencing a null object.
NumberFormatException	caused when a conversion between strings and number fails.
OutOfMemoryException	caused when there is not enough memory to allocate
StringIndexOutOfBoundException	caused when a program attempts to access a non-existent character position in a string.

Exceptions in java can be of two types:

Checked Exceptions :

- Handled explicitly in the code itself with the help of try catch block.
- Extended from java.lang.Exception class

Unchecked Exceptions :

- Not essentially handled in the program code, instead JVM handles such exceptions.
- Extended from java.lang.RuntimeException class

TRY AND CATCH

Try keyword is used to preface a block of code that is likely to cause an error condition and “throw” an exception. A catch block defined by the keyword catch “catches” the exception “thrown” by the try block and handles it appropriately.

A code can have more than one catch statement in the catch block, when exception in try block is generated, multiple catch statements are treated like cases in a switch statement.

Using Try and Catch for Exception Handling

```
class Error
{
	public static void main(String args[])
	{
		int a [] = {5,10};
		int b = 5;

		try
		{
			int x = a[2]/b-a[1];
		}
		catch(ArithmeticException e)
		{
			System.out.println("Division by zero");
		}
		catch(ArrayIndexOutOfBoundsException e)
		{
			System.out.println("ArrayIndexError");
		}
		catch(ArrayStoreException e)
		{
			System.out.println("Wrong data type");
		}

		int y = a[1]/a[0];
		System.out.println("y = " + y);
	}
}
```

FINALLY

Finally statement: used to handle exceptions that is not caught by any of the previous catch statements. A finally block in guaranteed to execute, regardless of whether or not an exception is thrown.

We can edit the above program and add the following finally block.

```
finally
        {
                int y = a[1]/a[0];
        System.out.println("y = " + y);
        }
```

THROWING YOUR OWN EXCEPTION

Own exceptions can be defined using throw keyword.

throw new Throwable subclass;

```
/* Throwing our own Exception */

import java.lang.Exception;
class MyException extends Exception
{
        MyException(String message)
        {
                super(message);
        }
}

class TestMyException
{
        public static void main(String args[])
        {
                int x = 5 , y = 1000;
                try
                {
                        float z = (float) x / (float) y ;
                        if(z < 0.01)
                        {
                                throw new MyException("Number is too small");
```

```
                    }
                }
                catch (MyException e)
                {
                        System.out.println("Caught my exception ");
                        System.out.println(e.getMessage());
                }
                finally
                {
                        System.out.println("I am always here");
                }
        }
}
```

MANAGING FILES IN JAVA

Storing data in variables and arrays poses the following problems:

- **Temporary Storage:** The data is lost when variable goes out of scope or when program is terminates.
- **Large data:** It is difficult

Such problems can be solved by storing data on secondary devices using the concept of files. Collection of related records stored in a particular area on the disk, termed as file. The files store and manage data by the concept of file handling.

Files processing includes:

- Creating files
- Updating files
- Manipulation of data

Java provides many features in file management like :

- Reading/writing of data can be done at the byte level or at character or fields depending upon the requirement.
- It also provides capability read/write objects directly.

STREAMS

Java uses concept of streams to represent ordered sequence of data, which is a path along which data flows. It has a source and a destination.

Streams are classified into two basic types :

- **Input Stream:** which extracts i.e. reads data from source file and sends it to the program.
- **Output Stream:** which takes the data from the program and send i.e writes to the destination.

STREAM CLASSES

They are contained in java.lang.io package.

They are categorized into two groups

1. **Byte Stream Classes:** provides support for handling I/O operation on bytes.
2. **Character Stream Classes:** provides support for managing I/O operations on characters.

BYTE STREAM CLASSES

Designed to provide functionality for creating and manipulating streams and files for reading/writing bytes.

Since streams are unidirectional there are two kinds of byte stream classes :

- Input Stream Classes
- Output Stream Classes

INPUT STREAM CLASSES

They are used to read 8-bit bytes include a super class known as InputStream. InputStream is an abstract class and defines the methods for input functions such as :

Method	Description
read( )	Reads a byte from input stream
read(byte b [ ])	Reads an array of bytes into b
read(byte b [ ], int n, int m)	Reads m bytes into b starting from the nth byte of b
available( )	Tells number of bytes available in the input
skip(n)	Skips over n bytes from the input stream
reset ( )	Goes back to the beginning of the stream
close ( )	Closes the input stream

OUTPUT STREAM CLASSES

These classes are derived from the base class OutputStream. OutputStream is an abstract class and defines the methods for output functions such as :

Method	Description
write( )	Writes a byte to the output stream
write(byte b[ ])	Writes all the bytes in the array b to the output stream
write(byte b[ ], int n, int m)	Writes m bytes from array b starting from the nth byte
close( )	Closes the output stream
flush( )	Flushes the output stream

READING/WRITING BYTES

Two common subclasses used are FileInputStream and FileOutputStream that handle 8-bit bytes.
FileOutputStream is used for writing bytes to a file as demonstrated below:

```
// Writing bytes to a file

import java.io.*;

class WriteBytes
{
        public static void main(String args[])
        {
                bytes cities [] = {'C','A','L','I','F','O','R','N','I','A',  '\n',
'V','E','G','A','S','\n','R','E','N','O','\n'};

                //Create output file stream
                FileOutputStream  outfile = null;
                try
                {
                        //connect the outfile stream to "city.txt"
                        outfile = new FileOutputStream("city.txt");
                        //Write data to the stream
                        outfile.write(cities);
                        outfile.close();
                }
                catch(IOException ioe)
                {
                        System.out.println(ioe);
```

```
                        System.exit(-1);
                }
        }
}
```

FileIntputStream is used for reading bytes from a file as demonstrated below:

```
//Reading bytes from a file

import java.io.*;

class ReadBytes
{
        public static void main(String args[])
        {
                //Create an input file stream
                FileInputStream  infile = null;
                int b;
                try
                {
                        //connect the infile stream to required file
                        infile = new FileInputStream(args [ 0 ]);

                        //Read and display
                        while( (b = infile.read ( ) ) !=-1)
                        {
                                System.out.print((char) b );
                        }
                        infile.close();
                }
                catch(IOException ioe)
                {
                        System.out.println(ioe);
                        System.exit(-1);
                }
        }
}
```

CHARACTER STREAM CLASSES

Two kinds of character stream classes :

Reader Stream Classes

- Designed to read character from the files.
- Class Reader is the base class for all other classes.
- These classes are similar to input stream classes except their fundamental unit of information, while reader stream uses characters.

Writer Stream Classes

- Performs all output operations on files.
- Writes characters
- The Writer class is an abstract class which is the base class, having methods identical to those of OutputStream.

READING/WRITING CHARACTERS

The two subclasses of Reader and Writer classes for handling characters in files are FileReader and FileWriter.

```
// Copying characters from one file to another

import java.io.*;

class CopyCharacters
{
        public static void main (String args[])
        {
                //Declare and create input and output files
                File inFile = new File("input.dat");
                File outFile = new File("output.dat");
                FileReader ins = null;                          //creates file stream
ins
                FileWriter outs = null;                         //creates file stream
outs

                try
                {
                        ins = new FileReader(inFile);           //opens inFile
                        outs = new FileWriter(outFile);                 //opens outFile

                        //Read and write
                        int ch;
                        while((ch = ins.read( ))!=-1)
                        {
```

```
                    outs.write(ch);
                }
            }
            catch(IOException e)
            {
                System.out.println(e);
                System.exit(-1);
            }
            finally
            {
                try
                {
                    ins.close();
                    outs.close();
                }
                catch (IOException e)
                {}
            }
        }
}
```

JAVA COLLECTIONS

The collections framework contained in the java.util package defines a set of interfaces and their implementations to manipulate collections, which serve as a container for a group of objects.

INTERFACES

Collection framework contains many interfaces such as Collection, Map and Iterator.
The interfaces and their description are mentioned below:

Interface	Description
Collection	collection of elements
List (extends Collection)	sequence of elements
Queue (extends Collection)	special type of list
Set (extends Collection)	collection of unique elements
SortedSet (extends Set)	sorted collection of unique elements
Map	collection of key and value pairs, which must be unique

SortedMap (extends Map)	sorted collection of key value pairs
Iterator	object used to traverse through a collection
List (extends Iterator)	object used to traverse through a sequence

CLASSES

The classes available in the collection framework implement the collection interface and sub-interfaces. They also implement Map and Iterator interfaces.

Classes and their Corresponding interfaces are listed :

Class	Interface
AbstractCollection	Collection
AbstarctList	List
Abstract	Queue
AbstractSequentialList	List
LinkedList	List
ArrayList	List, Cloneable and Serializable
AbstractSet	Set
EnumSet	Set
HashSet	Set
PriorityQueue	Queue
TreeSet	Set
Vector	List, Cloneable and Serializable
Stack	List, Cloneable and Serializable
Hashtable	Map, Cloneable and Serializable

Array List Implementation

```
// Using the methods of array list class

import java.util.*;
class Num
{
        public static void main(String args[])
        {
                ArrayList num = new ArrayList ();
                num.add(9);
```

```
                num.add(12);
                num.add(10);
                num.add(16);
                num.add(6);
                num.add(8);
                num.add(56);

                //printing array list
                System.out.println("Elements : ");
                num.forEach((s) -> System.out.println(s));

                //getting size
                System.out.println("Size of array list is: ");
                num.size();

                //retrieving specific element
                int n = (Integer) num.get(2);
                System.out.println(n);

                //removing an element
                num.remove(4);

                //printing array list
                System.out.println("Elements : ");
                num.forEach((s) -> System.out.println(s));
        }
}
```

Linked List Implementation

```
import java.util.Scanner;
class LinkedList
{
        public static void main (String args[])
        {
        Scanner s = new Scanner(System.in);
                List list = new List();
                System.out.println("Enter the number of elements you want to enter in LL
: ");
                int num_elements = s.nextInt();
```

```
                int x;
                for(int i =0;i<=num_elements;i++)
                {
                        System.out.println("Enter element : ");
                        x = s.nextInt();
                        list.insert(x);
                }
                System.out.println(">>>>> LINKED LIST AFTER INSERTION IS  : ");
                list.print();
                int size = list.count();
                System.out.println(">>>>> SIZE OF LL => "+size);
                System.out.println("Enter the node to be inserted in the middle: ");
                int mid_element = s.nextInt();
                list.insertMiddle(mid_element);
                System.out.println(">>>> LL AFTER INSERTING THE NEW ELEMENT
IN THE MIDDLE ");
                list.print();
        }
}
```

HashSet Implementation

```
import java.util.*;
class HashSetExample
{
        public static void main(String args[])
        {
                HashSet hs = new HashSet();
                hs.add("D");
                hs.add("W");
                hs.add("G");
                hs.add("L");
                hs.add("Y");
                System.out.println("The elements available in the hash set are :" + hs);
        }
}
```

Tree Set Implementation

```
import java.util.*;
class TreeSetExample
{
        public static void main(String args[])
```

```
        {
                TreeSet ts = new TreeSet();
                ts.add("D");
                ts.add("W");
                ts.add("G");
                ts.add("L");
                ts.add("Y");
                System.out.println("The elements available in the hash set are :" + ts);
        }
}
```

Vector Class Implementation

```
import java.util.*;
class VectorExample
{
        public static void main(String args[])
        {
                Vector fruits = new Vector ();
                fruits.add("Apple");
                fruits.add("Orange");
                fruits.add("Grapes");
                fruits.add("Pineapple");
                Iterator it = fruits.iterator();
                while (it.hasNext())
                {
                        System.out.println(it.next);
                }
        }
}
```

Stack Class Implementation

```
import java.util.*;

public class StackExample
{
        public static void main (String args[])
        {
                Stack st = new Stack ();
                st.push("Java");
                st.push("Classes");
```

```
                st.push("Objects");
                st.push("Multithreading");
                st.push("Programming");

                System.out.println("The elements in the Stack : " + st);
                System.out.println("The elements at the top of Stack : " + st.peek());
                System.out.println("The elements popped out of the Stack : " + st.pop());
                System.out.println("The elements in the Stack after pop of the element : "
+ st);
                System.out.println("The result of search : " + st.search ("r e"));
        }
}
```

HashTable Class Implementation

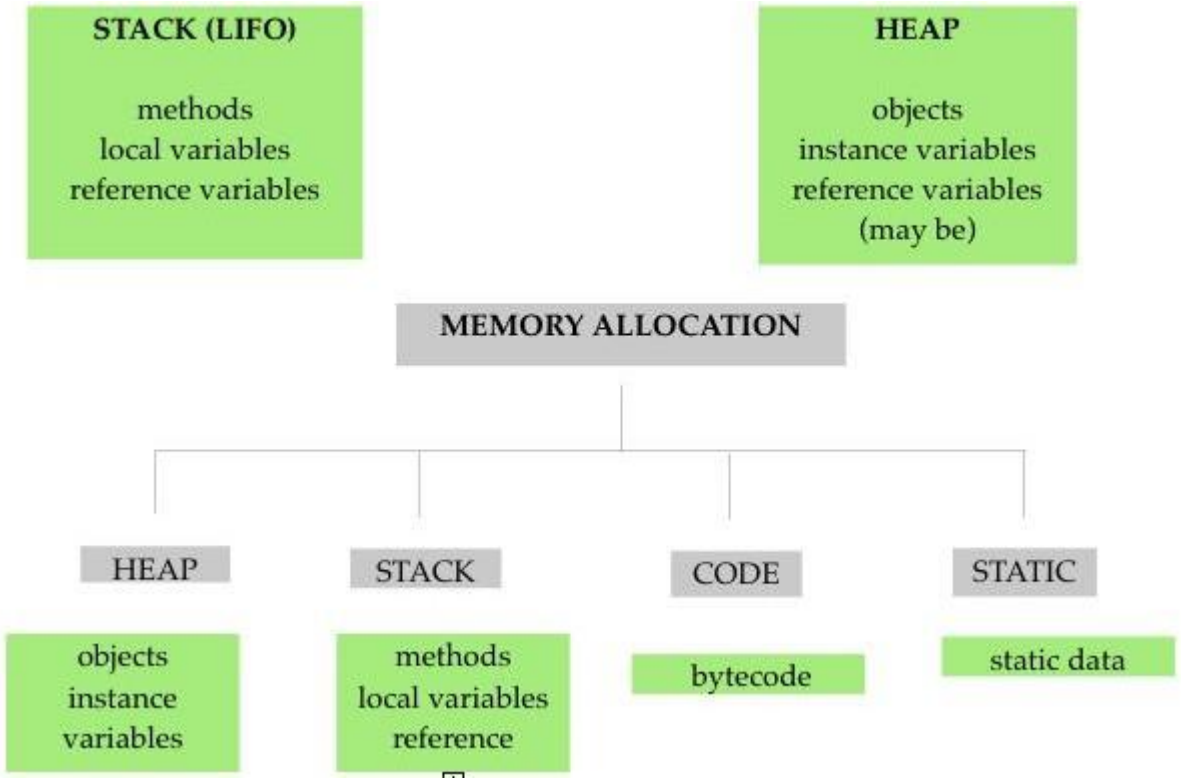
```
import java.util.*;

public class HashTableExample
{
        public static void main (String args[])
        {
                Hashtable ht = new Hashtable();
                ht.put("Item 1","Apple");
                ht.put("Item 2","Orange");
                ht.put("Item 3","Grapes");
                ht.put("Item 4","Pine");
                ht.put("Item 5","Kiwi");

                Enumeration e = ht.keys();
                while(e.hasMoreElements())
                {
                        String str = (String) e.nextElement();
                        System.out.println(ht.get(str));
                }
        }
}
```

MEMORY MANAGEMENT IN JAVA

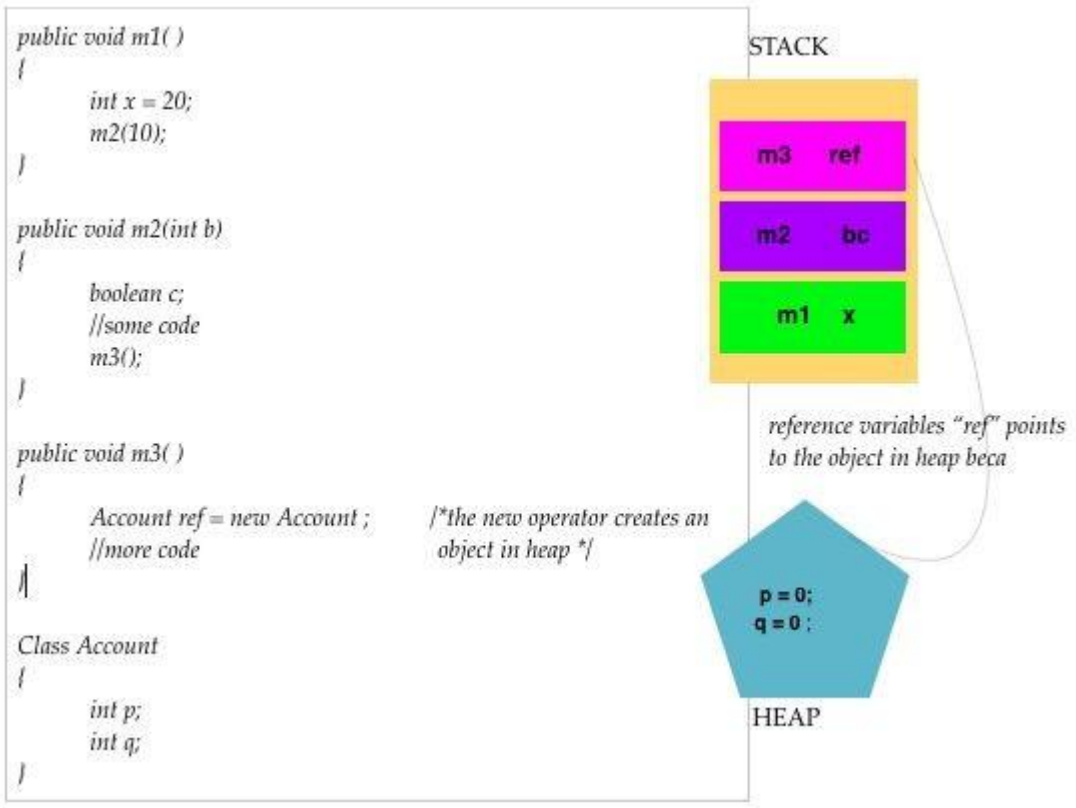
Memory is a collection of data represented in the binary format.

Memory management is :

- Process of allocating new objects
- Properly removing unused objects(garbage collection)



Example Illustrating Memory Management



1. When a method is called , frame is created on the top of the stack.
2. Once a method as completed execution, the flow of control returns to the calling method and its corresponding stack frame is flushed.
3. Local variables are created in stack.
4. Instance variables are created in the heap and are part of the object they belong to.
5. Reference variable are created in stack.

SOME COMMON JAVA CODING QUESTIONS

1. Enter radius and print diameter, perimeter and area

```
import java.util.Scanner;

class Circle
{
        public static void main (String args [])
        {
                double r,dia,peri,area  ;
                System.out.println("Enter the radius of circle : ");
```

```
                Scanner s = new Scanner (System.in);
                r = s.nextDouble();

                dia = 2*r;
                peri = 2*Math.PI*r;
                area = Math.PI*r*r;

                System.out.printf("The dia of circle is : %.2f \n", dia);
                System.out.printf("The peri of circle is : %.2f \n", peri);
                System.out.printf("The area of the circle is : %.2f \n", area);
        }
}
```

2. Print all the even numbers between x and y.

```
import java.util.Scanner;
class EvenOdd
{
        public static void main (String args[])
        {
                int x,y;
                Scanner s = new Scanner (System.in);
                System.out.println("Enter the values x , y : ");
                x = s.nextInt();
                y = s.nextInt();

                System.out.println(" **** EVEN NUMBERS BETWEEN GIVEN RANGE
ARE **** >> ");
                int count = x;
                while(count <=y)
                {
                        if(count % 2 == 0)
                        {
                                System.out.println(count);
                        }
                        count ++;
                }
        }
}
```

3. To check if given number is prime

```
import java.util.Scanner;

class Prime
{
	public static void main (String args[])
	{
		double num;
		int n;
		boolean isPrime = true;
		Scanner s = new Scanner(System.in);

		System.out.println("Enter the number to check :");
		num=s.nextDouble();

		n = (int) Math.sqrt(num);

		for(int i=2;i<=n;i++)
		{
			if(num % i == 0)
			{
				isPrime = false;
			}
			else
			{
				isPrime = true;
			}
		}

		if(isPrime)
		{
			System.out.println("***** NUMBER IS PRIME !!!! ****** ");
		}
		else
		{
			System.out.println("***** NUMBER IS NOT PRIME !!!! ****** ");

		}
	}
}
```

4. To check if the entered number is Palindrome

```
import java.util.Scanner;

class Palindrome
{
	public static void main(String args[])
	{
		int num,reverse=0,mode;
		Scanner s = new Scanner(System.in);

		System.out.println("Enter a number to check for Palindrome: ");
		num = s.nextInt();

		int number = num;
		while(num!=0)
		{
			//System.out.println(" number entering =  "+num);
			mode = num % 10;
			//System.out.println(" mode = "+mode);
			reverse =(reverse * 10 )+ mode;
			//System.out.println(" reverse = "+reverse);
			num = num/10;
			//System.out.println(" new num = "+num);
		}

		//System.out.println(" reverse out = "+reverse);

		if(reverse == number)
		{
			System.out.println(" **** PALINDROME !!! **** ");
		}
		else
		{
			System.out.println(" **** NOT A PALINDROME !!! **** ");
		}
	}
}
```

5. Pattern printing

```
*
* *
```

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

```
import java.util.Scanner;

class TriStars
{
        public static void main(String args[])
        {
                for(int i=0;i<=5;i++)
                {
                        for(int j=0;j<i;j++)
                                {
                                        System.out.print(" * ");
                                }
                        System.out.println();
                }
                System.out.println();
        }
}
```