Piyush Mahajan

1.Can we write try and finally without catch block what is the use??

Yes,  we can have try without catch block by using finally block.

You can use try with finally. As you know finally block always executes even if you have exception or return statement in try block.

public class TryWithoutCatchMain {

public static void main(String args[])

{

  try

  {

   System.out.println("Executing try block");

  }

  finally

  {

   System.out.println("Executing finally block");

  }

}

}

Output: Executing try block

Executing finally block

If you have return statement in try block, still finally block executes.

public class TryWithoutCatchMain {

public static void main(String args[])

{

  System.out.println(print());

}

public static String print()

{

  try

  {

   System.out.println("Executing try block");

   return "Return from try block";

  }

  finally

  {

   System.out.println("Executing finally block");

  }

}

}

Executing try block

Executing finally block

Return from try block

2.Can we create class as final?

The final modifier for finalizing the implementations of classes, methods, and variables.

The main purpose of using a class being declared as final is to prevent the class from being subclassed. If a class is marked as final then no class can inherit any feature from the final class.

You cannot extend a final class. If you try it gives you a compile time error.

You can declare some or all of a class's methods *final*. You use the final keyword in a method declaration to indicate that the method cannot be overridden by subclasses.

Methods called from constructors should generally be declared final. If a constructor calls a non-final method, a subclass may redefine that method with surprising or undesirable results.

Note that you can also declare an entire class final. A class that is declared final cannot be subclassed. This is particularly useful, for example, when creating an immutable class like the String class.

3.Why java is platform independent??

Java is a platform-independent language. In other words, you can write your code once and then run it anywhere, on any platform that provides the environment to run it.

This environment is the Java Virtual Machine (JVM). The JVM should be present to execute the code. The JVM is different for each platform.

In the case of Java, platform independence does not mean that you can run the code anywhere; you can run it wherever the environment is provided. This is the key point of platform independence in Java.

In Java, programs are compiled into byte code and that byte code is platform-independent. The byte code is executed by the Java Virtual Machine and the Java Virtual Machine is platform dependent. Java is platform-independent. Any machine to execute the byte code needs the Java Virtual Machine.

For example, Java can be use in Windows , Linux as well as mac operating system. Thus it is said that java is platform independent.

4.What is Metaspace and Heap memory??

Metaspace

Metaspace is a new memory space – starting from the Java 8 version; it has replaced the older permgen memory space. The most significant difference is how it handles memory allocation.

Specifically, this native memory region grows automatically by default.

We also have new flags to tune the memory:

* Metaspacesize and maxmetaspacesize – we can set the Metaspace upper bounds.
* Minmetaspacefreeratio – is the minimum percentage of class metadata capacity free after [garbage collection](https://www.baeldung.com/jvm-garbage-collectors)
* Maxmetaspacefreeratio – is the maximum percentage of class metadata capacity free after a garbage collection to avoid a reduction in the amount of space

Additionally, the garbage collection process also gains some benefits from this change. The garbage collector now automatically triggers the cleaning of the dead classes once the class metadata usage reaches its maximum metaspace size.

Therefore, with this improvement, JVM reduces the chance to get the outofmemory error.

Heap memory

Heap memory is a part of memory allocated to JVM, which is shared by all executing threads in the application. It is the part of JVM in which all class instances and are allocated. It is created on the Start-up process of JVM. It does not need to be contiguous, and its size can be static or dynamic. Space allocated to the memory is reclaimed through an automatic memory management process called garbage collection. Heap memory is a shared area that is utilized during the runtime of Java applications. It is created during the instantiation of Java Virtual Machine (JVM).

This memory is shared by instances of all the classes created during the runtime of an application. As per system configuration, the size of heap memory may be fixed or variable. In order to reclaim the space of heap memory, an automatic memory management process called garbage collection is triggered by JVM. JVM provides control to developers to vary the size of heap memory according to requirement.

**College Management**

**package** com.pack;

**import** java.util.Scanner;

**public** **class** CollegeManagement {

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

Scanner sc = **new** Scanner(System.***in***);

**int** std\_id;

**char** choice;

String std\_name;

**int** m\_hin, m\_eng, m\_mat, m\_sci, m\_soc;

System.***out***.println("Welcome to College Management");

System.***out***.println("Enter student id");

std\_id = sc.nextInt();

System.***out***.println("Enter student name");

std\_name = sc.next();

System.***out***.println("Enter marks in hindi ");

m\_hin = sc.nextInt();

System.***out***.println("Enter marks in english ");

m\_eng = sc.nextInt();

System.***out***.println("Enter marks in science");

m\_sci = sc.nextInt();

System.***out***.println("Enter marks in maths");

m\_mat = sc.nextInt();

System.***out***.println("Enter marks in social ");

m\_soc = sc.nextInt();

System.***out***.println("Student added successfully");

**do** {

System.***out***.println("11: Check student result");

System.***out***.println("22: Add student result");

System.***out***.println("33: Exit");

**int** option;

option = sc.nextInt();

**switch** (option) {

**case** 11: {

**int** stdid;

System.***out***.println("Enter student id to check result");

stdid = sc.nextInt();

**if** (stdid != std\_id) {

System.***out***.println("student id not found");

} **else** {

**double** percentage;

**int** total;

String result;

total = m\_hin + m\_eng + m\_mat + m\_sci + m\_soc;

percentage = total \* 100 / 500;

**if** (total >= 250) {

result = "pass";

} **else** {

result = "fail";

}

System.***out***.println("Student Result {id =" + std\_id + ", name =" + std\_name

+ ", marks=Subject Marks{hindi = " + m\_hin + ", english =" + m\_eng + ", maths =" + m\_mat

+ ", Science =" + m\_sci + ", social =" + m\_soc + "}, result =" + result + ", total ="

+ total + ", percentage =" + percentage + "}");

}

}

**break**;

**case** 22: {

**double** percentage;

**int** total;

String result;

total = m\_hin + m\_eng + m\_mat + m\_sci + m\_soc;

System.***out***.println("total =" + total);

percentage = total \* 100 / 500;

System.***out***.println("percentage =" + percentage);

**if** (total >= 250) {

System.***out***.println("pass");

} **else** {

System.***out***.println("fail");

}

}

**break**;

**case** 33: {

System.***out***.println("Exit");

}

**break**;

**default**: {

System.***out***.println("wrong option");

}

}

System.***out***.println("Do you want to continue (Y/N):");

choice = sc.next().charAt(0);

} **while** (choice == 'y' || choice == 'Y');

System.***out***.println("Bye");

}

}

Character count

**package** com.pack;

**import** java.util.Scanner;

**public** **class** CharacterCount {

**static** **final** **int** ***MAX\_CHAR*** = 256;

**static** **void** getOccuringChar(String str) {

**int** count[] = **new** **int**[***MAX\_CHAR***];

**for** (**int** i = 0; i < str.length(); i++)

count[str.charAt(i)]++;

**char** ch[] = **new** **char**[str.length()];

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

ch[i] = str.charAt(i);

**int** f = 0;

**for** (**int** j = 0; j <= i; j++) {

**if** (str.charAt(i) == ch[j])

f++;

}

**if** (f == 1)

System.***out***.println(str.charAt(i) + " :" + count[str.charAt(i)]);

}

}

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

Scanner sc = **new** Scanner(System.***in***);

String s;

System.***out***.println("Enter the string");

s = sc.next();

*getOccuringChar*(s);

}

}

Remove Duplicates

package com.pack;

import java.util.ArrayList;

import java.util.Arrays;

public class RemoveDuplicate {

public static <T> ArrayList<T> deleteDuplicates(ArrayList<T> list) {

ArrayList<T> newList = new ArrayList<T>();

for (T element : list) {

if (!newList.contains(element)) {

newList.add(element);

}

}

return newList;

}

public static void main(String[] args) {

ArrayList<Integer> list = new ArrayList<>(Arrays.asList(1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1, 2, 5, 7, 6, 3, 2));

System.out.println("ArrayList without removing duplicates is ->" + list);

ArrayList<Integer> newList = deleteDuplicates(list);

System.out.println("ArrayList after deletion of duplicates is ->" + newList);

}

}

**Strings Equal**

**package** com.pack;

**import** java.util.Scanner;

**public** **class** StringsEqual {

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

Scanner sc = **new** Scanner(System.***in***);

System.***out***.println("Enter the first string: ");

String string1 = sc.nextLine();

System.***out***.println("Enter the second string :");

String string2 = sc.nextLine();

System.***out***.println("Are both strings same: ");

**if** (string1.equals(string2) == **true**) {

System.***out***.println("Yes");

} **else** {

System.***out***.println("No");

}

}

}

**Multiples Using Recursion**

**package** com.pack;

**public** **class** MultiplesUsingRecursion {

**static** **void** multiples(**int** i, **int** n) {

**if** (i >= 1 || n <= 1) {

System.***out***.println("Enter Number : " + i);

}

**while** (i <= 1 || n <= 1)

;

System.***out***.println("Multiples of " + i + " are:");

**for** (**int** counter = 0; counter < n; counter++) {

System.***out***.println(i \* (2 + counter) + " ");

}

}

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

**int** s = 2;

*multiples*(s, 9);

}

}

Highest Salary

**package** com.pack;

**public** **class** HighestSalary {

**private** **long** id;

**private** String name;

**private** **int** salary;

**public** HighestSalary(**long** id, String name, **int** salary) {

**this**.id = id;

**this**.name = name;

**this**.salary = salary;

}

**public** **int** getSalary() {

**return** salary;

}

**public** **void** setSalary(**int** salary) {

**this**.salary = salary;

}

@Override

**public** String toString() {

**return** "Employee [id=" + id + "," + " name=" + name + "," + " salary=" + salary + "]";

}

}

package com.pack;

import java.util.Arrays;

import java.util.List;

class Employee {

public static void main(String[] args) {

List<HighestSalary> employees = Arrays.asList(new HighestSalary(1, "Piyush", 50000), new HighestSalary(2, "Omkar", 30000),

new HighestSalary(3, "Yash", 40000) , new HighestSalary(4, "Pratik", 10000), new HighestSalary(5, "Pramod", 60000));

int maxSalary = employees.stream().map(HighestSalary::getSalary).max(Integer::compare).get();

System.out.println("Max salary of the employee:" + maxSalary);

System.out.print("Employee details:");

employees.stream().filter(emp -> emp.getSalary() == maxSalary).forEach(System.out::println);

}

}

Menu

**package** com.pack;

**import** java.sql.Connection;

**import** java.sql.DriverManager;

**import** java.sql.PreparedStatement;

**import** java.sql.ResultSet;

**import** java.sql.SQLException;

**import** java.util.Scanner;

**public** **class** Menu {

**public** **static** **void** main(String[] args) **throws** ClassNotFoundException, SQLException {

Class.*forName*("org.apache.derby.jdbc.ClientDriver");// loading drivers

Connection conn = DriverManager.*getConnection*("jdbc:derby://localhost:1527/training;create=false", "derby",

"derby");

**char** choice;

Scanner sc = **new** Scanner(System.***in***);

**int** menu;

**do** {

System.***out***.println("1: View Employee Data ");

System.***out***.println("2: Update Employee Data ");

System.***out***.println("3: Add Employee ");

System.***out***.println("4: Delete Employee Data ");

System.***out***.println("5: View Employee Data for everyone ");

System.***out***.println("6: Exit");

menu = sc.nextInt();

**switch** (menu) {

**case** 1: {

PreparedStatement st = conn.prepareStatement("select \* from app.employee where emp\_id=?");

System.***out***.println("Enter employee id to search data: ");

**int** id = sc.nextInt();

st.setInt(1, id);

ResultSet rs = st.executeQuery();

**while** (rs.next()) {

System.***out***.println("EMP\_ID = " + rs.getInt(1) + " EMP\_NAME = " + rs.getString(2) + " EMP\_SAL = "

+ rs.getInt(3));

}

}

**break**;

**case** 2: {

System.***out***.println("Enter employee id where we need to update data :");

**int** idu = sc.nextInt();

System.***out***.println("Enter Name = ");

String nameu = sc.next();

System.***out***.println("Enter Salary =");

**int** salaryu = sc.nextInt();

PreparedStatement stri = conn

.prepareStatement("update app.employee set emp\_nm=? , emp\_sal =? where emp\_id=?");

stri.setString(1, nameu);// execute the statement

stri.setInt(2, salaryu);

stri.setInt(3, idu);

stri.executeUpdate();

System.***out***.println("records updated");

}

**break**;

**case** 3: {

PreparedStatement str = conn.prepareStatement("insert into app.employee values(?,?,?)");

System.***out***.println("Enter Employee Data :");

System.***out***.println("Employee Id = ");

**int** insertid = sc.nextInt();

System.***out***.println("Employee Name = ");

String insertname = sc.next();

System.***out***.println("Employee Salary = ");

**int** insertsalary = sc.nextInt();

str.setInt(1, insertid);

str.setString(2, insertname);

str.setInt(3, insertsalary);// execute the statement

str.executeUpdate();

System.***out***.println(" records inserted");

}

**break**;

**case** 4: {

PreparedStatement strin = conn.prepareStatement("delete from app.employee where emp\_id=? ");

System.***out***.println("Enter employee id to delete data: ");

**int** empdel = sc.nextInt();

strin.setInt(1, empdel);

// strin.setString(2, "java");// execute the statement

strin.executeUpdate();

System.***out***.println("records deleted");

}

**break**;

**case** 5: {

PreparedStatement st4 = conn.prepareStatement("select \* from app.employee");

ResultSet rs4 = st4.executeQuery();

**while** (rs4.next()) {

System.***out***.println(

"ID: " + rs4.getString(1) + " Name: " + rs4.getString(2) + " Salary: " + rs4.getString(3));

}

}

**break**;

**case** 6: {

System.***out***.println("Exit");

}

**break**;

**default**: {

System.***out***.println("wrong option");

}

}

System.***out***.println("Do you want to continue (Y/N):");

choice = sc.next().charAt(0);

} **while** (choice == 'y' || choice == 'Y');

System.***out***.println("End");

}

}

1. What is garbage collector and how it works??

Garbage collector:

Garbage collection in Java is the process by which Java programs perform automatic memory management. Java programs compile to bytecode that can be run on a Java Virtual Machine, or JVM for short. When Java programs run on the JVM, objects are created on the heap, which is a portion of memory dedicated to the program. Eventually, some objects will no longer be needed. The garbage collector finds these unused objects and deletes them to free up memory.

In Java, the programmer need not care for all those objects which are no longer in use. Garbage collector destroys these objects. The main objective of Garbage Collector is to free heap memory by destroying unreachable objects.

How it works:

Java garbage collection is an automatic process. Automatic garbage collection is the process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects. An in-use object, or a referenced object, means that some part of your program still maintains a pointer to that object. An unused or unreferenced object is no longer referenced by any part of your program. So the memory used by an unreferenced object can be reclaimed. The programmer does not need to mark objects to be deleted explicitly. The garbage collection implementation lives in the JVM.

2. What is Heap Space??

Heap memory is a part of memory allocated to JVM, which is shared by all executing threads in the application. It is the part of JVM in which all class instances and are allocated. It is created on the Start-up process of JVM. It does not need to be contiguous, and its size can be static or dynamic. Space allocated to the memory is reclaimed through an automatic memory management process called garbage collection. Heap memory is a shared area that is utilized during the runtime of Java applications. It is created during the instantiation of Java Virtual Machine (JVM).

This memory is shared by instances of all the classes created during the runtime of an application. As per system configuration, the size of heap memory may be fixed or variable. In order to reclaim the space of heap memory, an automatic memory management process called garbage collection is triggered by JVM. JVM provides control to developers to vary the size of heap memory according to requirement.

3. What is Metaspace??

Metaspace is a new memory space – starting from the Java 8 version; it has replaced the older permgen memory space. The most significant difference is how it handles memory allocation.

Specifically, this native memory region grows automatically by default.

We also have new flags to tune the memory:

* Metaspacesize and maxmetaspacesize – we can set the Metaspace upper bounds.
* Minmetaspacefreeratio – is the minimum percentage of class metadata capacity free after [garbage collection](https://www.baeldung.com/jvm-garbage-collectors)
* Maxmetaspacefreeratio – is the maximum percentage of class metadata capacity free after a garbage collection to avoid a reduction in the amount of space

Additionally, the garbage collection process also gains some benefits from this change. The garbage collector now automatically triggers the cleaning of the dead classes once the class metadata usage reaches its maximum metaspace size.

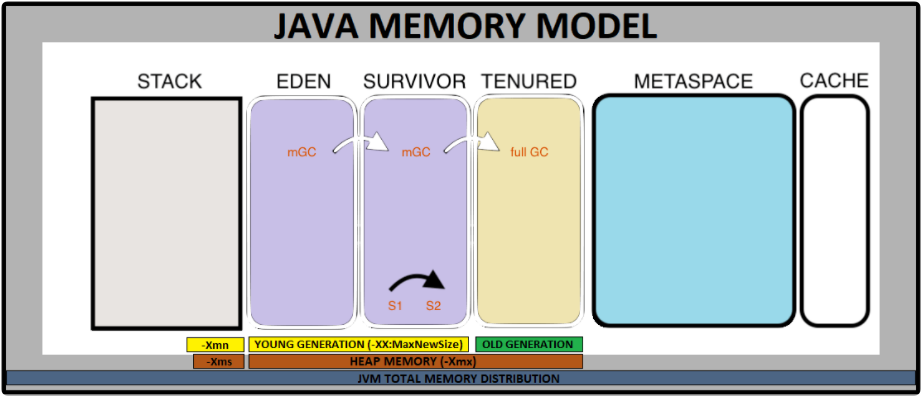
Therefore, with this improvement, JVM reduces the chance to get the outofmemory error.

4. What is Java Memory Model??

The Java memory model specifies how the Java virtual machine works with the computer's memory (RAM). The Java virtual machine is a model of a whole computer so this model naturally includes a memory model - AKA the Java memory model.

It is very important to understand the Java memory model if you want to design correctly behaving concurrent programs. The Java memory model specifies how and when different threads can see values written to shared variables by other threads, and how to synchronize access to shared variables when necessary.

The original Java memory model was insufficient, so the Java memory model was revised in Java 1.5. This version of the Java memory model is still in use in Java today (Java 14+).



In the image shown above for Java Memory Model:

Heap Space: Eden + Survivor + Tenured

Non-Heap Space: Stack + MetaSpace + Reserved (Not shown here)

Cache

5. What is Young and Old generations in Garbage Collector??

In a typical application most objects are very short-lived. On the other hand, some objects last for a very long time and even until the application is terminated. When using generational garbage collection, the heap area is divided into two areas—a young generation and an old generation—that are garbage-collected via separate strategies.

Objects are usually created in the young area. Once an object has survived a couple of GC cycles it is tenured to the old generation. After the application has completed its initial startup phase, most allocated objects will not survive their first or second GC cycle. The number of live objects that need to be considered in each cycle should be stable and relatively small.

Allocations in the old generation should be infrequent, and in an ideal world would not happen at all after the initial startup phase. If the old generation is not growing and therefore not running out of space, it requires no garbage-collection at all. There will be unreachable objects in the old generation, but as long as the memory is not needed, there is no reason to reclaim it.

To make this generational approach work, the young generation must be big enough to ensure that all temporary objects die there. Since the number of temporary objects in most applications depends on the current application load, the optimal young generation size is load-related. Therefore, sizing the young generation, known as generation-sizing, is the key to achieving peak load.

Unfortunately, it is often not possible to reach an optimal state where all objects die in the young generation, and so the old generation will often often require a concurrent garbage collector. Concurrent garbage collection together with a minimally growing old generation ensures that the unavoidable, stop-the-world events will at least be very short and predictable.

On the other hand, while there is a very high number of allocations in the young generation at the beginning of each GC cycle, there is only a small portion of objects alive after each GC cycle. This leads to a high level of fragmentation using the GC strategies we have discussed so far. You might think that using free lists would be a good option, but this will slow down allocations. An alternative strategy of executing a full compaction every time has a negative effect on pause time. Instead, most JVMs implement a strategy in the young generation, known as copy collection.

6. What is Eden and Survivor space??

**Eden Space:**

When we create an object, the memory will be allocated from the Eden Space.

**Survivor Space:**

This contains the objects that have survived from the Young garbage collection or Minor garbage collection. We have two equally divided survivor spaces called S0 and S1.