**AWS**

1. **Characteristics of Cloud:**

* **On demand self service**
* **Broad Network access**
* **Scalability**
* **Resource pulling**
* **Measured services**

1. **Reserved Words:**

In Java total 53 Reserved word are available among them **goto** and **const** are unused

* Keywords (50):
* Used Keywords (48)
* **Reserved words for Data Types** (8): **byte** (1 byte, 0), **short** (2 bytes, 0), **int** (4 byte, 0), **long** (8 bytes, 0L – if we will not wite L then it will be considered as integer and if the value will be out of range from int then it starts showing error then we will have to write l), **float** (4 bytes, 0.0 f - mandatory), **double** (8 bytes, 0.0d – writing d is optional), boolean and char(2 bytes).
* **Reserved words for flow control** (11): if, else, switch, case, default, while, do, for, break, continue, return
* **Reserved word for modifiers (11):** public, private, protected, final, abstract, static, synchronized, native, strictfp, transient, volatile
* **Reserved words for Exception handling (6)**: try, catch, finally, throw, throws, asserts
* **Class related keywords (6):** class, interface, package, import, extends, implements
* **Object related keywords (4):** new, instanceof, super, this
* **Void return type**

1. **Arrays:**

An Array is indexed collection of finite number of homogeneous data elements.

Syntax of array declaration: **int a[] = new int[10];**

* It is legal to have an Array with size 0
* If we are trying to specify Array size with negative int value then we will get Runtime exception saying **NegativeArrySizeException**
* The maximum allowed Array size in Java is **2147483647** which is the maximum value of int datatype
* Whenever we are trying to print any object reference internally **toString()** will be called which is by default implemented to return String in the following form:

**CalssName@hashcode\_in\_hexadecimal** from

1. **length vs length():**

length is final attribute/variable application for arrays and it represents size of array.

And length() is final method present in String class which is used to return size of String.

1. **Type of Variables:**

**Based on type of variable value, variable is divided into 2 types**

1. Primitive variable
2. Reference variable

**Based on purpose and position of declaration all variable are divided int 3 types:**

1. **Instance variables**

* If the value of a variable is varied from object to object such type of variable is called instance variable
* For every object a separate copy of instance variable will be created
* Instance variable will be created at the time of object creation and will be destroy at the time of Object distraction, Scope of instance variable is exactly same as scope of Object
* Instance variable will be stored in heap area
* Instance variable should be declared with in the class but outside of any methods, blocks or Constructors
* We cannot access instance variable directly from static area but can access by using object reference
* For instance, variables JVM will always provide default values and we are not required to perform initialization explicitly

1. **Static variables**

* If the value of variable is not varied from object to object, then it is not recommended to declare that variable as instance variable, we have to declare such type of variable at class level with **static** modifier
* In the case of instance variable for every object a separate copy will be created but in the case of a static variable a single copy will be created at class level and shared by all the objects of that class
* **Static variables** should be declared with in the class directly but outside of any method, block or constructor
* Static variable will be created at the time of class loading and destroyed at the time of class unloading hence the scope of a static variable is exactly same as the scope of the class
* Start JVM
* Create & start main thread
* Locate Test.class
* Load Test.class
* Execute main method
* Unload Test.class
* Terminate main thread
* Shutdown JVM
* **Static** **variable will be stored in method area**
* We can access static variable directly from both instance and static area
* For the static variable we are not required to perform initialisation explicitly JVM will always provide default value
* Static variable also known as class level variable for kids’ local variables

1. **Local variables**

* Some times to meet the temporary requirement of the programmers we can declare variable inside a methods, blocks or constructor such type of variables are called the **Local variables**, **Stack variables**, **Temporary variables** or **Automatic variables**
* **Local variable** will be stored inside the **stack memory**
* **Local variables** will be created while executing the block in which we declared it once the block execution completes automatically local variable will be destroyed hence the scope of Local variable is the same in which we declare it
* For Local variables JVM will not provide default values compulsory we have to perform initialization explicitly before using that variable
* If we are not using that variable then no need to initialize
* It is not recommended to initialize local variable inside logical blocks, because there will be no guarantee for the execution of that blocks
* It is highly recommended to initialize local variable before use at least with default value
* The only applicable modifier for local variable is **final** by mistake if we are trying to apply other modifier then we will get compile time error

**Note:**

* **Overloading concept** is not applicable for **Variables** in Java
* When you declared a variable of the same name in subclass, that’s called **Hiding**. The resulting Subclass will now actually have both properties, we can access the once from the super class with **“super.var”** or **((SuperClass)this).var**. The variable don’t even have to be of same type, they are just 2 variables storing the name much like two Overloaded methods
* If we are not declaring any access modifier then by default it is **default** but this rule is applicable only for instance and static variable not for Local variable
* For Instance, and Static variable JVM will always provide default values and we are not required to perform initialization explicitly but for Local variable JVM will not provide default value
* **For final variable if it is instance variable then we will have to initialize and in local final variable no need to initialize**

1. **Var-Args methods:**

* Before Java 1.4 version we cannot declare a method with variable number of arguments
* We can mix var-args parameters with normal parameters
* If we mix normal parameter with var-args parameter then var-args parameter should be last parameter eg: **void m1(char ch, String …s)**
* Inside var-args method we can take only one var-args parameter and we cannot take more than one var-args parameter
* Inside a class we cannot declare var-args method and corresponding one dentitional array method otherwise we will get Compile time error
* In case of method overloading var-args method get least priority
* Whenever one dentitional array present we can replace with var-args parameter
* Whenever var-args parameter present we cannot replace with one dimensional array

1. **Main method:**

* Weather class contains main method or not and is it declared properly or not that will be validate at Runtime not at compile time, If JVM will not find main method then Runtime exception will come saying **NoSuchMethodError: main**
* At runtime JVM will always searches for main method with the following prototype: **public static void main(String[] args)**
* **Public – To call by JVM from anywhere**
* **Static – without existing object, JVM has to call this method**
* **Void – main method won’t return anything to JVM**
* **Main – this is the name configured inside JVM**
* **String[] args – command line arguments**
* Even syntax of main method is very strict however we can do following changes:
* Instead of public static we can take static public
* We can declare String[] args as String args[], String []args, etc
* Instead of args we can take any valid java identifiers
* We can replace String[] with var-args parameter String… s
* Overloading of main method is possible however JVM will always call String[] args argument main method only, other method will be behave like normal method
* We can declare main method with – **final, synchronized, and strictfp**
* Overloading of main method is possible however JVM will always call String[] args argument main method only, other method will be behave like normal method
* Inheritance concept applicable for main method hence while executing child class if child does not have main method, then parent class main method will be executed
* From Java version 1.7 main method is mandatory to start the program

1. **Operators:**

* We can apply increment and decrement operators only for variables but not for constant values, if we are trying to do so we will get compile time error. **eg:** **int x = 10; int y = ++ 10** not allowed whereas **int x = 10; int y = ++ x;** allowed
* Nesting of increment and decrement operators are not allowed eg: **int x = 10; int y = ++ (++x)** not allowed
* For **final** variables we cannot apply increment and decrement operators
* We can apply this ++ and – for every primitive type variable except **boolean**
* **In Integral arithmetic (byte, short, int, long)** there is no way to represent **infinity** hence if infinity is result then we will get ArithmeticException. **Eg: /by Zero**
* But in floating point arithmetic float and double has a way to represent infinity for this Float & Doble classes contains the following two constants: **POSITIVE\_INFINITY** and **NEGATIVE\_INFINITY.** Even throughresult is infinity we will not get ArithmeticException
* In integral arithmetic there is no way to represent undefined results hence if the result is undefined, we will get RuntimeException saying ArithmeticException, however in floating point arithmetic there is a way to represent undefined that is NaN (Not a number)

**String Concatenation operator (+):**

* The only overloaded operator in Java is **+** operator sometimes it acts as arithmetic addition operator and sometimes it acts as string concatenation operator
* If at least one argument is of String type, then **+** operator acts as concatenation operator and if both argument is of number type, then **+** operator acts as arithmetic operator
* String a = “durga”; int b = 10; c = 20; d = 30;
* Sop(a+b+c+d); // durga102030
* Sop(b+c+d+a); // 60durga
* Sop(b+c+a+d); // 30durga30
* Sop(b+a+c+d); // 10durga2030

**Relational operator (<, <=, >, >=):**

* We can apply Relational operator with every Primitive type except boolean
* We cannot apply relational operators for object type Eg: sop(“durga” > “durga123”)
* Nesting of Relational operators are not allowed Eg: sop(10<10<10)

**Equality (=) operator:**

* We can apply equality operator for every Primitive type including boolean type also Eg: sop(false == false);
* We can use this operator for object type as well

**Note:**

* What is difference between == and .equals() method?

In general we can use == operator for reference comparation (address comparison) and .equals() method for content comparison however object class .equals() method meant of reference comparison only.

**instanceof operator:**

We can use instanceof operator to check whether the given Object is of instance of particular type or not

* To use instanceof operator compulsory there should be some relation between argument types either child to parent or parent to child otherwise we will get Compile time error, saying incompatible type
* For any class or interface X **null instanceof X** is always false, so in case of null it will return false so this null safe

**Short-circuit operator (&&, ||):**

These are exactly same as bitwise operators (&, |) except the following differences:

|  |  |
| --- | --- |
| **&, |** | **&&, ||** |
| Both arguments should be evaluated always | Second argument evaluation is optional |
| Relatively performance is low | Relatively performance is high |
| Applicable for both boolean and integral types | Applicable for boolean but not for integral types |

**Note:**

* x && y => y will be evaluated if x is true, i.e if x is false then then y will be not evaluated
* x || y => y will be evaluated if x is false also if x is true then y will not be evaluated

**Type-cast operator:**

There are 2 types of Type-casting:

1. **Implicit Type-Casting (automatically / widening / up-casting)**

* Compiler is responsible to perform this typecasting, whenever we are assigning smaller datatype value into bigger datatype variable implicit Type-casting will happen
* There is no loss of information in this type-casting

1. **Explicit Type-Casting**

* Programmer is responsible to perform explicit type-casting
* It is also known as narrowing or down casting
* Whenever we are assigning bigger datatypes value to smaller datatype variable then explicit type-casting required
* There may be a change of loss of information in this type-casting

**Conditional operator (? :):**

The only possible ternary operator in Java is conditional operator Eg: int x = (10<20) ? 30 : 40 ;

**new vs newInstance():**

**new is an operator** in Java which is used for create an object if we know class name at the beginning whereas newInstance() is a method present in class Class() which can also be used to create object if we don’t know class name at the beginning and it is available dynamically at run time.

* newInstance() method internally calls no-arg constructor hence to use newInstace() method compulsory corresponding class should contain no-argument constructor otherwise we will get compile time error saying instantiation error
* While using new operator at runtime if corresponding if the corresponding .class file is not available then we will get **NoClassDefFoundError** however in newInstace() we will get **ClassNotFoundException**.

**instanceof vs isInstance():**

* **Instanceof** is an operator in Java we can use instanceof to check wether the given object is of particular type or not and we know the type at the beginning
* isInstance() is a method present in java.lang.class we can use is isInstance() method to check whether the given Object is of particular type or not and we don’t know the type at the beginning and it is available dynamically at runtime

**switch case:**

* The allowed argument type for the switch statement are **byte, short, char, int, Byte, Short, Character, Integer, enum and String**
* Case and default are optional
* Inside switch every statement should be under some case or default means independent statement are not allowed inside switch otherwise we will get compile time error
* Every case should be constant expression
* Every case label should be in the range of switch argument type otherwise we will get compile time error
* Duplicate case labels are not allowed
* Within the switch if any case is matched from that case onward all the statement will be executed util break or end of the switch this call fall-through inside switch
* Default case will be executed if nothing matched

**Package**

Package is the group of related classes, interfaces and enums it promotes name space.

* In any Java source file, there can be at most one package statement means more than one package statement is not allowed otherwise we will get compile time error
* In any Program the first non-comment statement should be Package statement otherwise we will get compile time error

1. **Class level Modifiers:**

* The only applicable modifiers for top level classes are – **public, <default>, final, abstract, strictfp**

1. **Access specifier vs Access Modifiers:**

public, private, protected, default is considered as access specifier except these remaining are considered as modifiers

* **Public:** If a member declares as public then we can access that member from anywhere however corresponding class should be visible, before checking visibility we have to check class visibility
* **Default:** If a member declares as default then we can access that member only with in the current package from outside of package we can’t access
* **Private:** If a member is private then we can access that member only within the class from outside of class we cannot access
* **Protected:** If a member declares as protected then we can access that member only with in the current package and only in child classes of outside package **protected = <default> + kids**
* **private < default < protected < public**

1. **final modifier:**

Final (**final**) is modifier applicable for classes, method and variables

* **final class:**
* If we declare any class as final then we cannot create sub class or child class of that class
* Every method present inside a final class is always final by default, but every variable present in final class need not to be final
* **final method:**
* If we declare any method as final then we cannot override it
* **final variable:**
* If we declare any variable as final then we cannot reassign the value of that variable
* If we declare any reference variable as final then also, we cannot reassign the value of that reference variable

1. **abstract modifier:**

**abstract** is modifier applicable for classes and methods but not for variables

* **abstract class:**
* If we declare any class as abstract then we are not allowed to create object of that class (Because of partial implementation)
* If a class contains at least one abstract method then compulsory we should declare class as abstract otherwise we will get compile time error (Reason – If a class contains at least one abstract method then implementation is not complete and hence it is not recommended to create object to restrict object instantiation compulsory we should declare class as abstract)
* Even through class doesn’t contains any abstract method still we can declare class as abstract means abstract class can contain 0 number of abstract methods
* **abstract method:**
* Even though we do not know about implementation still we can declare a method as abstract i.e for abstract method only declaration is available but not implementation, so abstract method declaration should ends with **;**
* Child class is responsible to provide implementation for parent class abstract methods
* By declaring abstract method in the parent class, we can provide guideline to the child class such that which method compulsory child has to implement
* abstract method never talks about implementation if any modifier talks about implementation, then it forms illegal combination with abstract modifier

1. **final vs abstract:**

* abstract method compulsory we should override in the child classes to provide implementation whereas we cannot override final class
* For final classes we cannot create child class whereas for abstract classes we should create child class to provide implementation, hence final abstract combination is illegal for classes
* abstract class can contain final method whereas final class cannot contain abstract method
* It is highly recommended to use abstract modifier because it promotes several OOPS features like inheritance and polymorphism ========= starts from page 114 strictfp

1. **Strictfp modifier: [Strict floating point]**

* We can use strictfp with for classes and methods but not for variables
* Usually, the result of floating-point arithmetic is varied from platform to platform if we want platform independent result for floating point arithmetic then we should go for strictfp modifier.
* If a method declared as strictfp all floating-point calculations in that method has to follow **IEEE 754** standard so that we will get platform independent result.
* If a class declare as strictfp then every floating-point calculation present in every concreate method has to follow **IEEE 754** standard, so that we will get platform independent result.
* Abstract modifier never talks about implementation whereas strictfp method always talks about implementation hence abstract and strictfp combination is illegal for methods.

1. **Synchronized modifier:**

Synchronized (synchronized) is the modifier applicable for methods and blocks but not for classes and variables.

If multiple threads are trying to operate simultaneously on the same java object, then there may be a chance of data inconsistency problem this is called “race condition” we can overcome this problem by using synchronized keywords.

If a method or block declared as synchronized then at a time only one thread is allowed to execute that method or block on the given object.

1. **Native modifier:**

Native (native) is the modifier applicable only for methods. The methods which are implemented in non-java mostly in C or C++ are called native methods or foreign methods.

The main objective of native keyword are:

* To improve performance of the system
* To achieve machine level or memory level communication
* T use already existing legacy non-java code

1. **Transient keyword:**

Transient (transient) is the modifier applicable only for variables, we use this keyword in serialization context.

* At the time of serialization if we don’t want to save the value of a particular variable to meet security constraint then we should declare that variable as transient
* At the time of serialization JVM ignores original value of transient variable and save default value to the file hence transient means not to serialize
* **In hibernate we use @Transient to ignore that attribute into the database**

1. **Volatile keyword:**

Volatile (volatile) is the modifier applicable only for variable.

* If the value of variable keeps on changing by multiple Threads, then there may be a change of data inconsistency problem, we can solve this problem by using volatile modifier if a variable declares as volatile then for every thread JVM will create a separate local copy every modification performed by the thread will takes place in local copy, so that there is no effect on the remaining threads.
* Final variable means value never changes where as volatile variable means the value keeps on changing hence volatile final is illegal combination

1. **Interface:**

* Interface is the contract between service provider and client or Interface provides the communication between two or more objects of different classes.
* In Interface we define only methods means what client expects like in ATM machine all the main functions show on screen which is interface
* From client point of view an interface defines the set of services what he is expecting and from Service provider point of view an interface defines the set of services what he is offering. Hence only contract between client and service provider is considered as an interface.
* **Inside interface all variables are by default final, public and static (not private) and all methods are always public and abstract weather we declared or not hence interface considered as 100% pure abstract class.**
* A class can extends only one class at a time where as an interface can extends any number of interfaces simultaneously

1. **Interface naming conflicts:**

* If two interfaces contain a method with same signature and same return type then in the implementation class, we have to provide implementation for only one method
* If two interfaces contain a method with same name but different argument types then in the implementation class, we have provided implementation for both methods and these methods acts as overloaded methods
* If two interfaces contain a method with same signature but different return types then it is impossible to implement both interfaces simultaneously

1. **Variable naming conflicts:**

* Two interfaces can contain a variable with the same name and there may be a chance of variable naming conflicts but we can solve this problem by using interface names

1. **Marker Interface:**

* If an interface doesn’t contain any method and by implementing that interface if our class objects will get some ability such type of interfaces are called **Marker interface** or **Ability interface** or **Tag interface**
* Without having any methods how the objects will get some ability in marker interfaces – Internally JVM is responsible to provide required ability
* Is it possible to create our own marker interface? – Yes, but customization of JVM is required

1. **Adapter Classes:**

Adapter classes is a simple java class that implements an interface with only empty implementation

1. **Interface vs Abstract class vs Concreate class:**

* If we don’t know anything about implementation just, we have requirement specification then we should go for interface
* If we are talking about implementation but not completely then we should go for abstract class
* If we are talking about implementation completely and ready to provide service then we should go for concreate class

1. **Difference between Interface and Abstract class:**

|  |  |
| --- | --- |
| **Interface** | **Abstract class** |
| * Inside interface every method is always public and abstract weather we are declaring of not, hence interface is considered as 100% abstract | * Every method present inside abstract class need not be public and abstract and we can declare concreate method as well |
| * Every variable present inside interface is always public static final weather we are declaring of not | * Every variable present in abstract class need not to be public and static and final |
| * Inside interface we cannot declare static and instance blocks | * Inside Abstract we can declare static and instance blocks |
| * Inside interface we can not declare constructor | * Inside abstract class we can declare constructor |
| * If we don’t know anything about implementation and just, we have requirement specification then we can go for interface | * If we are taking about implementation but not completely then we should go for Abstract class |

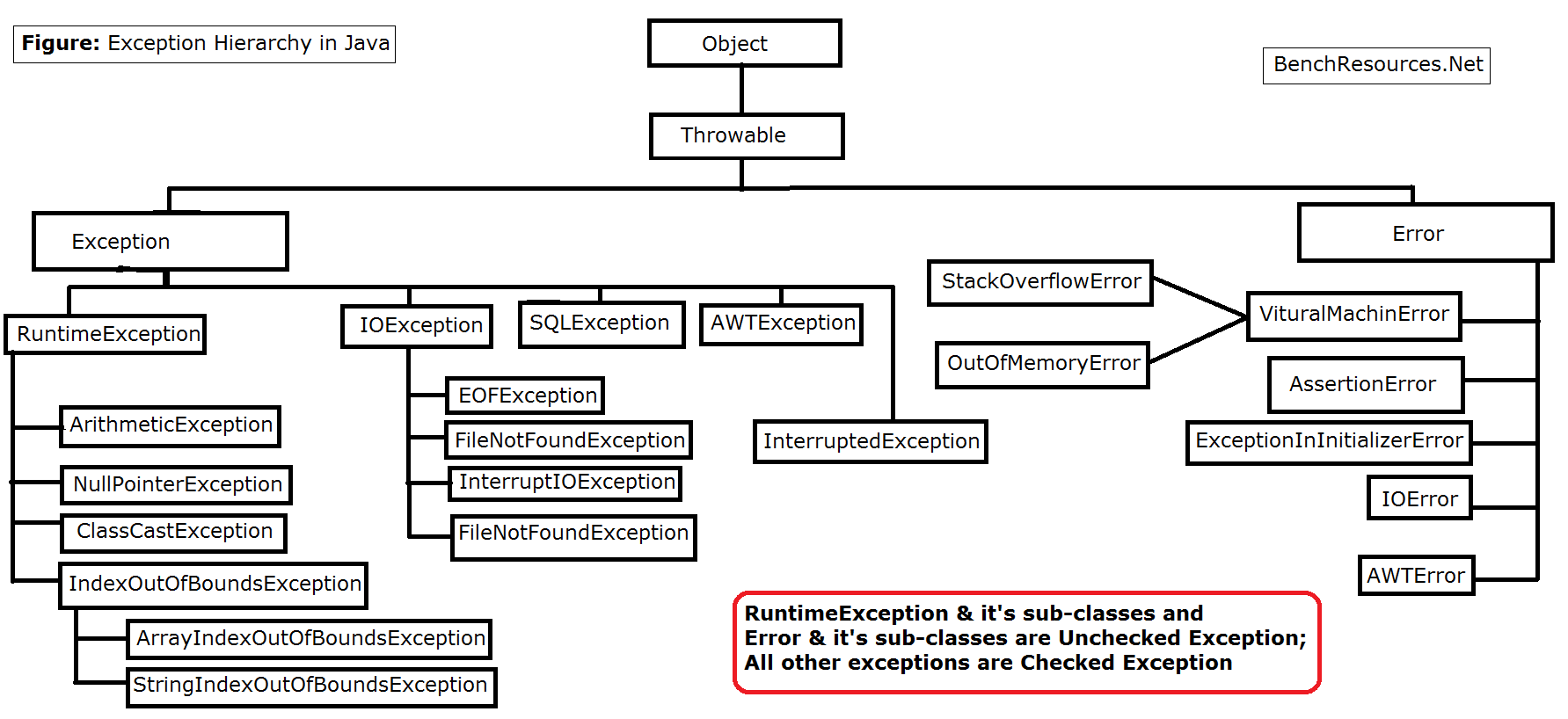
1. Anyway, we cannot create object for Abstract class but Abstract class can contain constructor what is the need? – Abstract class constructor will be executed whenever we are creating child class object to perform initialization of child class object.

**Exception**

An unwanted unexpected event that disturbs normal flow of the program is called Exception.

1. **Exception Hierarchy:**

* **Throwable** class act as base class for Exception hierarchy, it is root for all Exception and Errors
* Throwable class defines two child classes
* **Exception** – Most of the cases Exception will be raised because of our program logic and these are recoverable
* **Error** – Most of the cases enums won’t be caused because of our program logic, these are due lack of system resources



1. **Checked and Unchecked Exception:**

* **Checked Exception** – The exception which are checked by the compiler for smooth execution of the program at runtime are called checked exception. Eg – FileNotFoundException, InterruptedException etc.
* **Unchecked Exception** – The exception which are not checked by the compiler weather programmer is handling or not are called unchecked exception. Eg – ArithmeticException, NullPointerException, ClassCastException etc.
* **RuntimeException and its child class, Error and its child classes are unchecked exception**

1. **Fully checked and Partially checked Exception:**

* **Fully Checked –** A checked exception is said to be fully checked if an only if its all-child classes are checked. Eg – IOException, InterruptedException
* **Partially checked –** A checked exception is said to be partially checked exception if some of its child classes are unchecked. Eg – Exception, Throwable
* The only partially checked exception in entire exception hierarchy are **Exception** and **Throwable**

1. **Finally block:**

* It is not recommended to maintain clean-up code inside a try or catch block because there is no guaranty for execution of every statement with in the try or catch block, hence we require a special palace which should be executed always irrespective of exception raised or not raised, handled or not handled to maintain clean-up code so this place called as Finally (finally) block
* Even thorough return statement is present in try or catch block still finally block will execute, after that only return will be considered by JVM
* Default exception handler can handle only one exception at a time and it is the most recently raised exception
* There is only one situation where finally will not execute and that is when JVM encounters System.exit(0), on this statement call JVM itself get shutdown
* System.exit(0) – The argument 0 means normal termination and non-zero means abnormal termination

1. **Final vs finally vs finalize:**

* Final (final): - It is modifier applicable for classes methods and variables for more details find in previous pages
* Finally: - It is a block which is always associated with try and catch and which can be used to maintain clean-up code
* Finalize: - It is a method present in object class to maintain clean-up code just before destroying the object garbage collector calls finalize () method on that object to perform clean-up activity.

1. **Throw vs throws:**

|  |  |
| --- | --- |
| Throw | Throws |
| * Throws keyword is used to handover our own exception object to the JVM | * Throws keyword is used to delegate the exception handling responsibility to caller method |
| * After throw keyword we cannot use any other statement otherwise unreachable code exception (compile time) error will come | * Throws always used with method signature |
| * Always used with in the method | * Always used with method signature |
| * Used for both checked and unchecked exception | * Recommended to use with checked exception |
| * We use throw with instance | * Instance not required |
| * At a time only one we can throw only one exception | * At a time, we can throws multiple exception |

1. **Notes:**

* Whenever we are taking try with multiple catch blocks the order of catch block is very important and that order should be child to parent (top to bottom) otherwise we will get compile time error saying exception xxx has already seen caught.
* If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but it can declare unchecked exception.
* If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.
* If we want to create user define exception class as checked exception then we should extends **Exception** class and if want unchecked exception then we should extends our class from RuntimeException.
* Example of user define exception:

**public class ExceptionDemo extends Exception {  
   
 public ExceptionDemo(String message){  
 super(message);  
 }  
}**

1. **Exception handling Spring project**

We can handle exception at entire project level by using @RestControllerAdvice or @ControllerAdvice3