Exception Handling

What is an exception ?

- Run time error detected by JVM

- It is an alteration to the normal flow of execution

What is the need of exception handling ?

1. To continue with the program execution , even after run time errors(Eg . invalid inputs,Business Logic(B.L) failures,validation failures, file not found, invalid casting etc.)

2. To separate Business logic (in try block) from error handling logic(catch block)

3. To avoid un necessary checking with boolean conditions.

Flow of exception handling

Refer to a diagram - " day7_help\exception handling\exc-handling-flow.png"

Eg. In case of int division by 0

- JVM creates the instace of : ArithmeticExcpetion & throws the exception to the code.
- Keyword used - throw
- Typical syntax : throw Throwable instance;
- In above case it uses -
  throw new ArithmeticException("/ by 0");
- Then JVM checks for the **MATCHING** catch clause
- If it exists – JVM executes the catch block & then continues in the normal manner.
- If it doesn't exist - JVM supplies a default handler , which aborts the code , printing information , including exception name , error message & stack trace

NOTE -

Currently "**throw**" keyword : is used by JVM to raise a system | built in exception (eg : ArithmeticException, ArrayIndexOutOfBoundsException, NullPointerException,ClassCastException,IOException, SocketException, SQLException etc)

Programmer can also use the same “throw" keyword to raise a custom exception .

Inheritance hierarchy of exception handling classes

- Refer to the diagram : day7_help\exception handling\exc-inheritance-hierarchy.png

Keywords in exception handling

- Try , catch , finally , throw , throws , try-with-resources

try-catch syntax.

- try block is used to wrap the Business logic & catch block used for error handling logic
- try {....} catch (ArithmeticException e){..}
- try {....} catch (ArithmeticException e){..} catch (NullPointerException e) {..} catch(Exception e) {catch-all}
- try {....} catch (ArithmeticException | NullPointerException e){..} catch(Exception e) {....}

Checked vs. un checked exceptions

Java has created checked & unchecked exceptions differently because

Some errors(like unable to connect to DB , file read error) must be handled explicitly by programmer .

- These are considered as **checked** exceptions

Some other errors, are largely programming mistakes that need not clutter code with try-catch block .

- These are considered as **unchecked** exceptions.

Java compiler forces the handling of checked exceptions upon programmer and does not force the handling of unchecked exceptions.

Java Runtime System (JVM) doesn't differentiate between checked and un checked exceptions

(Meaning : If there is any un handled checked or un checked exception : JVM will abort code)

What are the ways of satisfying java compile in case of checked exceptions

- Actual handling using try-catch

OR

- use throws keyword :

- for delegating the exception handling to the caller .

throw vs throws (keywords used in exception handling)

throw

- keyword used to raise the exception(JVM uses it to throw system/built-in exception , Programmer uses it to throw custom exception).
- It appears in method definition.
- syntax : throw Throwable instance;

throws

- keyword meant for java compiler
- appears in method declaration
- Eg. public void show() throws IOException,InterruptedException{…}
- Meaning : show() : may throw the exception(indicates the possibility) & current method is NOT handling the exception/s
- It's caller should handle.
- Delegating the exception handling to the method caller.

When is adding "throws" keyword mandatory ?

- In case of unhandled checked exceptions.

- Otherwise , Javac error .

finally - a keyword in exception handling.

- It represents a block which is always executed.
- In case of no exception as well as exception or even before a method returns.
- It doesn't get called in case of JVM termination(System.exit(0)).
- Typical use case is for cleaning up of non java resources(Eg . File handle, socket , db connection, standard input)
- Syntax :
- try {....} catch(Exception e){...} finally {...} : legal
- try {....} finally {...} catch(Exception e){..} : illegal
- try {....} finally {...} : legal
- try {....} catch(ArithmeticException e){...} finally {...} :legal

try-with-resources syntax

- Syntax available since JDK 1.7
- Based on the interface - java.lang.AutoCloseable
- Method - public void close() throws Exception
- JVM automatically calls close() on the resources , opened in the try-with-resources block.
- Ensures prompt release of the resources.

syntax -

try(Create AutoCloseable object/s)

{.....} //JVM - calls close() auto matically on all these objects at the end.

Custom exceptions

- **Since JVM will not raise any exceptions for invalid inputs or Business logic failures , programmer has to create and raise user defined | custom exceptions.**

Development steps in creating and using Custom Exception

- Create a packaged , custom exception class , representing typically checked exception - by extending from Exception class
- Add parameterized constructor , to initialize error message.

  Eg. public class MyCustomException extends Exception {

  public MyCustomException (String mesg) {

  super(mesg); } }

- Create a separate class , for adding validation rules , for separation of concerns.
- Typically add static methods for validations, where in you can raise custom exceptions in case of validation failures.
- Use “throws” keyword to delegate the exception handling to the caller (i.e Main class) , for centralized exception handling.
- Invoke these validation methods from the main class.

String Handling

Refer to the diagram

- day7_help\string handling\string-overview.png
- In java.lang package , there are 3 classes for string handling
- String , StringBuilder and StringBuffer.
- String represents immutable char sequence.
- StringBuilder & StringBuffer represents mutable char sequence
- StringBuffer is a legacy class , guarantees synchronization (thread safety)
- StringBuilder is a modern class , without any guarantee of synchronization.
- In case of mutable char sequence , always prefer StringBuilder over StringBuffer.

Immutability of strings

- You cannot modify the contents of the String , once created (Immutable)
- String class API ,Eg. concat , toUpperCase , replace .. create actually a new String object in the heap.
- They are created as immutable for the purposes like
  - security , thread safety , caching (in string pool) , used as keys in HashMap

== vs equals method (reference equality vs. content equality)

- "==" is used for reference equality of 2 strings
- Not to be used in practical scenario

Regarding "equals"

- Object class has provided , **public boolean equals(Object o)** , for comparing 2 objects using their references(address)
- Meaning, the method returns true if & only if 2 references are referring to the same object in the heap , otherwise returns false.
- String class has overridden this method , to replace reference equality by contents equality of the char sequence in 2 strings (i.e based upon content equality) , in a case sensitive manner.

- In practical scenarios , always use equals method instead of "=="

Literal strings vs non literal string

Since Strings are used quite frequently in any java application and they are immutable , JVM uses string constant pool (SCP) or literal string pool , for the memory efficiency purpose.

How ?

When the java.lang.String class is loaded into Metaspace , the JVM also initializes a special data structure in the heap to act as the String Intern Pool | String constant Pool (SCP)

This pool is basically a JVM-managed HashTable/HashMap storing references to interned String objects.

- Initially pool is empty.

After Program execution begins , each string literal found , is resolved on first use.

JVM checks if an equivalent string(having the same contents) already exists in the String Pool.If not, it creates a new String object with that content and adds it(actually its reference) into the pool. If exists then simply returns the existing reference to the caller.All reference variables that use that literal will point to the same pooled object.

Programmer can also add the non literal strings in the pool , using

- **String's intern** method.
- public String intern();

- JVM checks if the pool already contains a string with the same contents
  If yes → returns the pooled reference.
  If no → Adds string into the pool & returns the pooled reference.

String class API (JDK 21) – Refer to java docs for complete details

1. Creation

```
String s1 = "hello";                // literal (SCP)

String s2 = new String("hi");// new object in heap

String s3 = s1.intern();          // pooled reference
```

2. Checking

```
s.equals(t)                  // content equality

s.equalsIgnoreCase(t)

s == t                        // reference equality

s.compareTo(t)              // lexicographic comparison (based upon unicode values of chars)

s.isEmpty()                 // length == 0

s.isBlank()                 // only whitespace
```

3. Querying

```
s.length()

s.charAt(2)

s.indexOf("lo")

s.lastIndexOf("l")

s.contains("he")

s.startsWith("He"), s.endsWith("lo")
```

4. Modification (results into creating new stringobject)

s.toUpperCase(), s.toLowerCase()

```
s.trim()                    // remove spaces
```

```
s.strip()                    // same as trim
s.stripLeading(), s.stripTrailing()
s.concat("abc")
s.replace("a","b")
s.replaceAll("\\d"," ") //creates a new string by replacing digits by space
s.substring(2,5)
s.repeat(3)                  // Java 11+
```

5. **SPLITTING & JOINING**

```
"a,b,c".split(",")              // ["a","b","c"]
String.join("-", "A","B","C") // "A-B-C"
```

6. **CONVERSION** |

```
String.valueOf(123)      // "123"
Integer.parseInt("123")// 123
s.toCharArray()              // char[]
s.getBytes()                 // byte[]
```

7. **FORMATTING**

```
String.format("Hi %s, age %d", "John", 25)
```