Collections, Exception Handling and I/O

Wrapper classes

The **wrapper class in Java** provides the mechanism *to convert primitive into object and object into primitive*.

autoboxing and **unboxing** feature convert primitives into objects and objects into primitives automatically. The automatic conversion of primitive into an object is known as autoboxing and vice-versa unboxing.

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

compareTo()

It compares two objects numerically and returns the result as -1, 0 or 1

parseInt()

It parses the String argument as a signed decimal Integer object.

parseFloat()

It parses the String argument as a Float object.

parseDouble()

It parses the String argument as a Double object.

parseLong()

It parses the String argument as a Long object.

parseByte()

It parses the String argument as a Byte object.

parseBoolean()

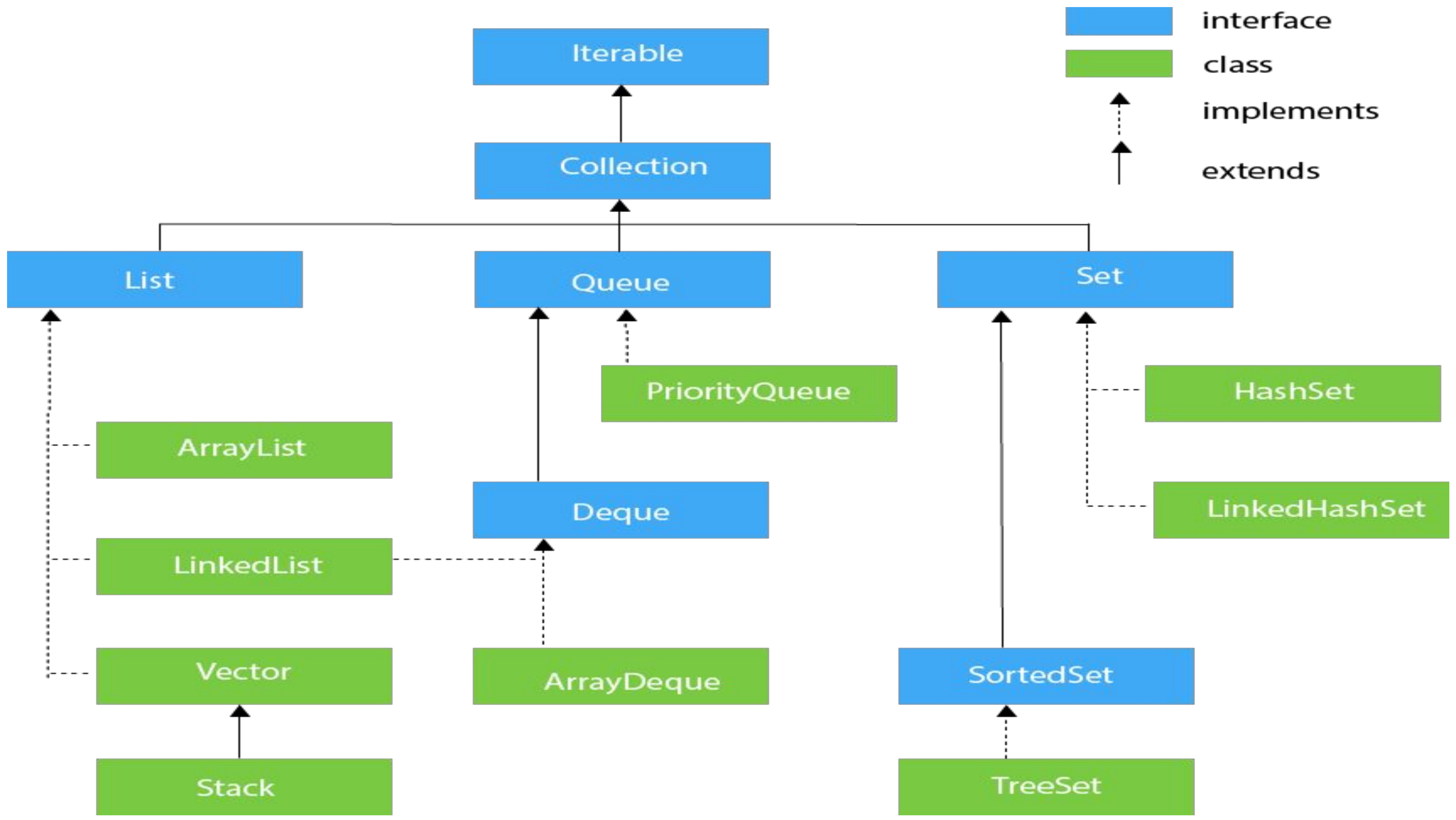
It parses the String argument as a Boolean object.

Collections in Java

- A Collection represents a single unit of objects, i.e., a group.
- The **Collection in Java** is a framework that provides an architecture to store and manipulate the group of objects.
- Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.
- Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

What is a framework?

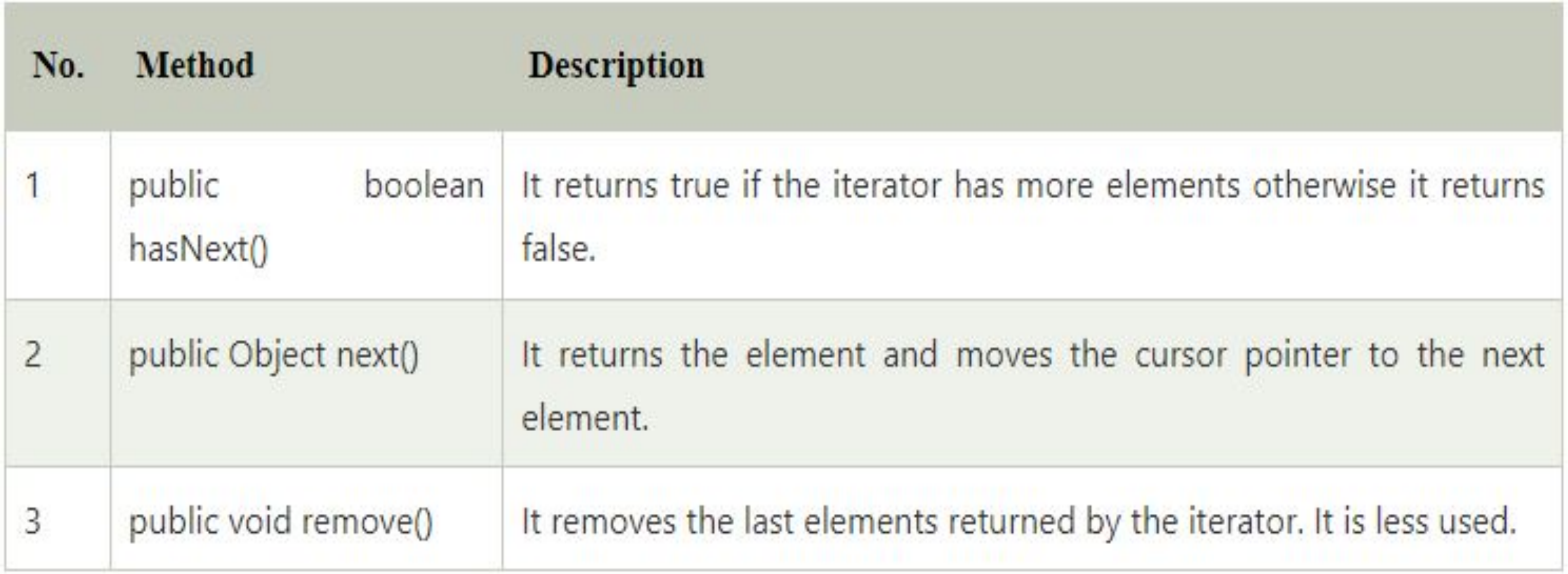
- It provides readymade architecture.
- It represents a set of classes and interfaces.
- It is optional.


interface
class
implements
extends
Iterable
Collection
List
Queue
Set
PriorityQueue
HashSet
ArrayList
Deque
LinkedHashSet
LinkedList
Vector
ArrayDeque
SortedSet
Stack
TreeSet

Iterator interface

Iterator interface provides the facility of iterating the elements in a forward direction only.

Methods of Iterator interface



No.	Method	Description
1	public boolean hasNext()	It returns true if the iterator has more elements otherwise it returns false.
2	public Object next()	It returns the element and moves the cursor pointer to the next element.
3	public void remove()	It removes the last elements returned by the iterator. It is less used.

Iterable Interface

The Iterable interface is the root interface for all the collection classes. The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.

It contains only one abstract method. i.e.,

```
Iterator<T> iterator()
```

List Interface

List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.

List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.

1. List <data-type> list1= new ArrayList();
2. List <data-type> list2 = new LinkedList();
3. List <data-type> list3 = new Vector();
4. List <data-type> list4 = new Stack();

ArrayList

- The ArrayList class implements the List interface.
- It is dynamic array i.e. It can grow and shrink dynamically.
- The ArrayList class maintains the insertion order and is non-synchronized.
- The elements stored in the ArrayList class can be randomly accessed.

Method	Description
void add(int index, E element)	It is used to insert the specified element at the specified position in a list.
boolean add(E e)	It is used to append the specified element at the end of a list.
boolean addAll(Collection<? extends E> c)	It is used to append all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.
boolean addAll(int index, Collection<? extends E> c)	It is used to append all the elements in the specified collection, starting at the specified position of the list.
void clear()	It is used to remove all of the elements from this list.
void ensureCapacity(int requiredCapacity)	It is used to enhance the capacity of an ArrayList instance.
E get(int index)	It is used to fetch the element from the particular position of the list.

E get(int index)	It is used to fetch the element from the particular position of the list.
boolean isEmpty()	It returns true if the list is empty, otherwise false.
Iterator()	
listIterator()	
int lastIndexOf(Object o)	It is used to return the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.
Object[] toArray()	It is used to return an array containing all of the elements in this list in the correct order.
<T> T[] toArray(T[] a)	It is used to return an array containing all of the elements in this list in the correct order.
Object clone()	It is used to return a shallow copy of an ArrayList.
boolean contains(Object o)	It returns true if the list contains the specified element.

int indexOf(Object o)	It is used to return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.
E remove(int index)	It is used to remove the element present at the specified position in the list.
boolean remove(Object o)	It is used to remove the first occurrence of the specified element.
boolean removeAll(Collection<?> c)	It is used to remove all the elements from the list.
boolean removeIf(Predicate<? super E> filter)	It is used to remove all the elements from the list that satisfies the given predicate.
protected void removeRange(int fromIndex, int toIndex)	It is used to remove all the elements lies within the given range.
void replaceAll(UnaryOperator<E> operator)	It is used to replace all the elements from the list with the specified element.

void retainAll(Collection<?> c)	It is used to retain all the elements in the list that are present in the specified collection.
E set(int index, E element)	It is used to replace the specified element in the list, present at the specified position.
void sort(Comparator<? super E> c)	It is used to sort the elements of the list on the basis of the specified comparator.
Spliterator<E> spliterator()	It is used to create a spliterator over the elements in a list.
List<E> subList(int fromIndex, int toIndex)	It is used to fetch all the elements that lies within the given range.
int size()	It is used to return the number of elements present in the list.
void trimToSize()	It is used to trim the capacity of this ArrayList instance to be the list's current size.

```
import java.util.ArrayList;

public class ArrayListExample {

  public static void main(String[] args) {

    ArrayList<String> fruits = new ArrayList<>();

    fruits.add("Apple");

    fruits.add("Banana");

    fruits.add("Cherry");

    fruits.add("Date");

    System.out.println("Fruits: " + fruits);

    String secondFruit = fruits.get(1);

    System.out.println("The second fruit is: " + secondFruit);

    System.out.print("Iterating over fruits: ");

    for (String fruit : fruits) {

      System.out.print(fruit + " ");

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

LinkedList

- LinkedList implements the Collection interface.
- It uses a doubly linked list internally to store the elements.
- It can store the duplicate elements. It maintains the insertion order and is not synchronized.
- In LinkedList, the manipulation is fast because no shifting is required.

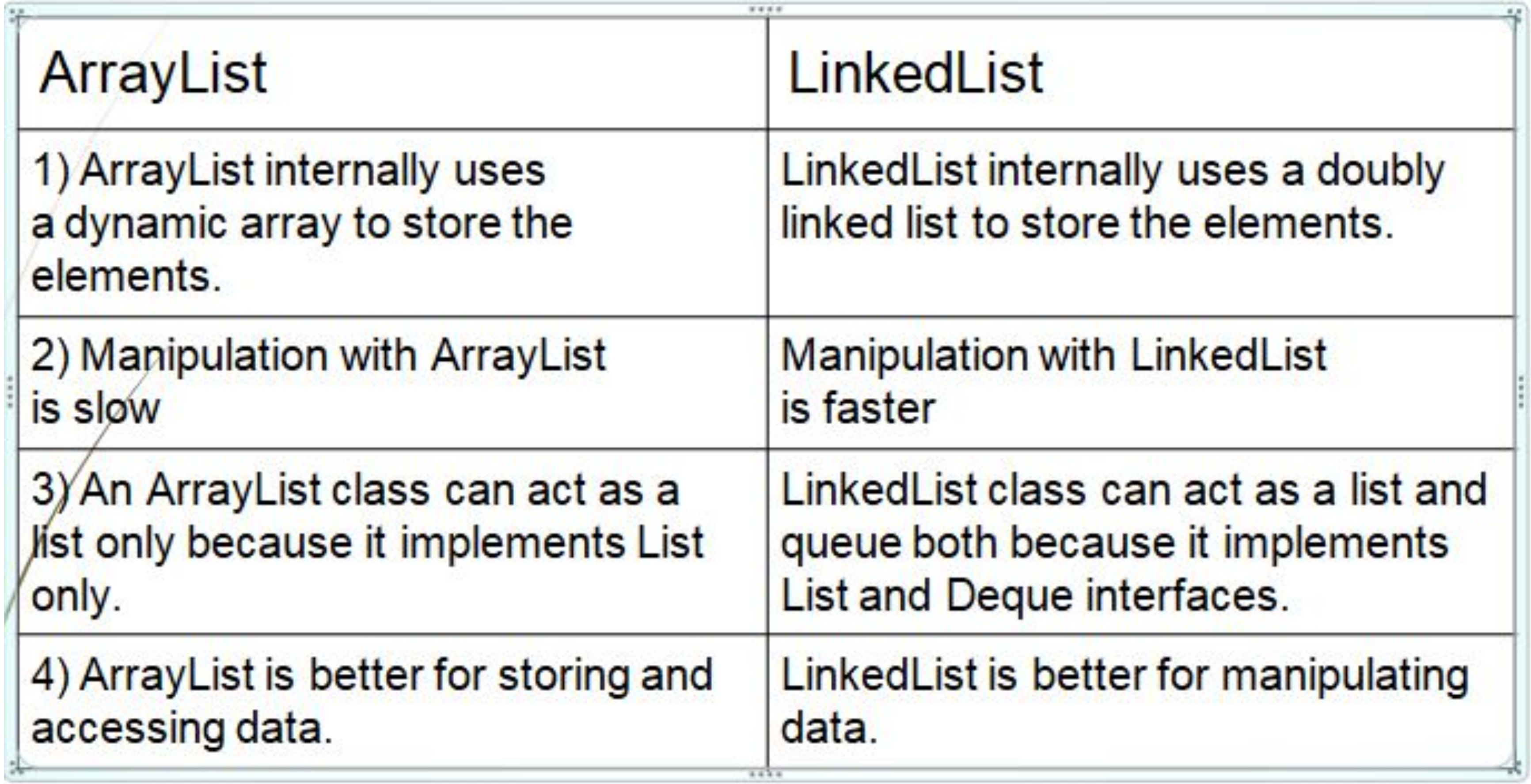
LinkedList methods:

- void addFirst(E e)
- void addLast(E e)
- Object getFirst()
- Object getLast()
- Object removeFirst()
- boolean removeFirstOccurrence(Object o)
- Object removeLast()
- boolean removeLastOccurrence(Object o)
- int indexOf(Object o)
- int lastIndexOf(Object o)

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


  LinkedList<String> al=new LinkedList<String>();
  al.add("Ravi");
  al.add("Vijay");
  al.add("Ravi");
  al.add("Ajay");
  Iterator<String> itr=al.iterator();
  while(itr.hasNext()){
   System.out.println(itr.next());
  }
 }
}
```

ArrayList vs LinkedList



ArrayList	LinkedList
1) ArrayList internally uses a dynamic array to store the elements.	LinkedList internally uses a doubly linked list to store the elements.
2) Manipulation with ArrayList is slow	Manipulation with LinkedList is faster
3) An ArrayList class can act as a list only because it implements List only.	LinkedList class can act as a list and queue both because it implements List and Deque interfaces.
4) ArrayList is better for storing and accessing data.	LinkedList is better for manipulating data.

Vector

- Vector uses a dynamic array to store the data elements.
- It is similar to ArrayList.
- However, It is synchronized and contains many methods that are not the part of Collection framework.

Vector Methods:

- addElement()
- capacity()
- elementAt(int index)
- elements()
- ensureCapacity()
- firstElement()
- insertElementAt(int index)
- iterator()
- removeAllElements()
- removeElement(Object o)
- removeElementAt(int index)
- setElementAt()

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

         Vector<String> vec = new Vector<String>();

        vec.add("Tiger");
        vec.add("Lion");
        vec.add("Dog");
        vec.add("Elephant");

        vec.addElement("Rat");
        vec.addElement("Cat");
        vec.addElement("Deer");
        System.out.println("Elements are: "+vec);
        }
}
```

Stack

- The stack is the subclass of Vector.
- It implements the last-in-first-out data structure, i.e., Stack.
- The stack contains all of the methods of Vector class and also provides its methods like boolean push(), boolean peek(), boolean push(object o), which defines its properties.

Method	Modifier and Type	Method Description
empty()	boolean	The method checks the stack is empty or not.
push(E item)	E	The method pushes (insert) an element onto the top of the stack.
pop()	E	The method removes an element from the top of the stack and returns the same element as the value of that function.
peek()	E	The method looks at the top element of the stack without removing it.
search(Object o)	int	The method searches the specified object and returns the position of the object.

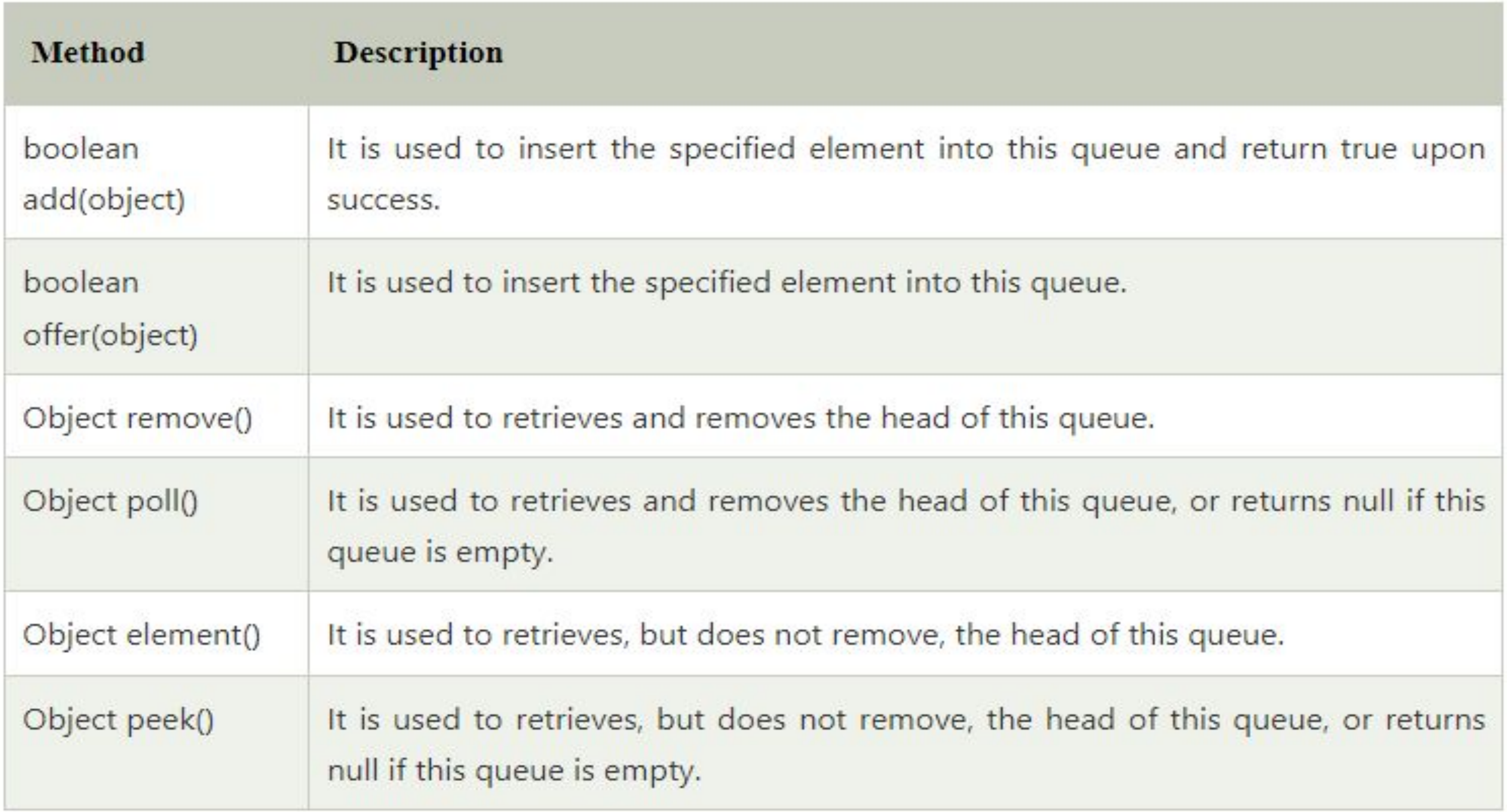
```
import java.util.*;
public class StackExample{
public static void main(String args[]){
Stack<String> stack = new Stack<String>();
stack.push("Ayush");
stack.push("Garvit");
stack.push("Amit");
stack.push("Ashish");
stack.push("Garima");
stack.pop();
Iterator<String> itr=stack.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Queue Interface

- Queue interface maintains the first-in-first-out order.
- It can be defined as an ordered list that is used to hold the elements which are about to be processed.
- There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.
- Queue<String> q1 = **new** PriorityQueue();
- Queue<String> q2 = **new** ArrayDeque();

PriorityQueue

- The PriorityQueue class implements the Queue interface.
- It holds the elements or objects which are to be processed by their priorities. PriorityQueue doesn't allow null values to be stored in the queue.



Method	Description
boolean add(object)	It is used to insert the specified element into this queue and return true upon success.
boolean offer(object)	It is used to insert the specified element into this queue.
Object remove()	It is used to retrieves and removes the head of this queue.
Object poll()	It is used to retrieves and removes the head of this queue, or returns null if this queue is empty.
Object element()	It is used to retrieves, but does not remove, the head of this queue.
Object peek()	It is used to retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.

```
import java.util.*;

public class TestJavaCollection5{

public static void main(String args[]){

PriorityQueue<String> queue=new PriorityQueue<String>();

queue.add("Amit Sharma");

queue.add("Vijay Raj");

queue.add("JaiShankar");

queue.add("Raj");

System.out.println("head:"+queue.element());

System.out.println("head:"+queue.peek());

System.out.println("iterating the queue elements:");
Iterator itr=queue.iterator();

while(itr.hasNext()){

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

}

queue.remove();

queue.poll();

System.out.println("after removing two elements:");

Iterator<String> itr2=queue.iterator();

while(itr2.hasNext()){

System.out.println(itr2.next());

}

}

}
```

Deque Interface

- Deque interface extends the Queue interface.
- In Deque, we can remove and add the elements from both the side.
- Deque stands for a double-ended queue which enables us to perform the operations at both the ends.
-
- Deque can be instantiated as:
    - Deque d = **new** ArrayDeque();

ArrayDeque

- ArrayDeque class implements the Deque interface.
- It facilitates us to use the Deque.
- Unlike queue, we can add or delete the elements from both the ends.
- ArrayDeque is faster than ArrayList and Stack and has no capacity restrictions.

Methods of ArrayDeque

add(Element e)
addAll(Collection<? extends E> c)
addFirst(Element e)
addLast(Element e)
clear()
clone()
contains(Obj)
element()
getFirst()
getLast()
isEmpty()
iterator()

offer(Element e)
offerFirst(Element e)
offerLast(Element e)
peek()
poll()
pop()
push(Element e)
remove()
remove(Object o)
removeAll(Collection<?> c)
removeFirst()
removeFirstOccurrence(Object o)

removeLast()

```
import java.util.*;
public class ArrayDequeExample{
public static void main(String[] args) {
Deque<String> deque = new ArrayDeque<String>();
deque.add("Gautam");
deque.add("Karan");
deque.add("Ajay");
for (String str : deque) {
System.out.println(str);
}
}
}
```

Set Interface

Set Interface in Java is present in java.util package. It extends the Collection interface. It represents the unordered set of elements which doesn't allow us to store the duplicate items. We can store at most one null value in Set. Set is implemented by HashSet, LinkedHashSet, and TreeSet.

Set can be instantiated as:

1. Set<data-type> s1 = **new** HashSet<data-type>();
2. Set<data-type> s2 = **new** LinkedHashSet<data-type>();
3. Set<data-type> s3 = **new** TreeSet<data-type>();

HashSet

- HashSet class implements Set Interface.
- It represents the collection that uses a hash table for storage.
- It contains unique items.

Methods of HashSet

add(E e)

clear()

contains(Object o)

remove(Object o)

isEmpty()

size()

clone()

```
import java.util.*;
public class HashsetExample{
public static void main(String args[]){
HashSet<String> set=new HashSet<String>();
set.add("Ravi");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
Iterator<String> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}  }
```

LinkedHashSet

- LinkedHashSet class represents the LinkedList implementation of Set Interface.
- It extends the HashSet class and implements Set interface.
- Like HashSet, It also contains unique elements.
- It maintains the insertion order and permits null elements.

```
import java.util.*;

class LinkedHashSet1{

 public static void main(String args[]){

        LinkedHashSet<String> set=new LinkedHashSet();

                set.add("One");

                set.add("Two");

                set.add("Three");

                set.add("Four");

                set.add("Five");

                Iterator<String> i=set.iterator();

                while(i.hasNext())

                {

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

                }  }}
```

SortedSet Interface

- SortedSet is the alternate of Set interface that provides a total ordering on its elements.
- The elements of the SortedSet are arranged in the increasing (ascending) order.
- The SortedSet provides the additional methods that inhibit the natural ordering of the elements.
- The SortedSet can be instantiated as:

```
SortedSet<data-type> set = new TreeSet();
```

TreeSet

- Java TreeSet class implements the Set interface that uses a tree for storage.
- Like HashSet, TreeSet also contains unique elements.
- However, the access and retrieval time of TreeSet is quite fast.
- The elements in TreeSet stored in ascending order.

Methods of TreeSet

add(Object o)
addAll(Collection c)
Comparator comparator()
contains(Object o)
descendingIterator()
descendingSet()
first()
floor(E e)
headSet(Object toElement)
higher(E e)
lower(E e)
pollFirst()
pollLast()
remove(Object o)

```
import java.util.*;
public class TreesetExample{
public static void main(String args[]){
TreeSet<String> set=new TreeSet<String>();
set.add("Ravi");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
Iterator<String> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

HashMap

Java **HashMap** class implements the Map interface which allows us *to store key and value pair*, where keys should be unique. If you try to insert the duplicate key, it will replace the element of the corresponding key. It is easy to perform operations using the key index like updation, deletion, etc. HashMap class is found in the java.util package.

- Java HashMap contains values based on the key.
- Java HashMap contains only unique keys.
- Java HashMap may have one null key and multiple null values.
- Java HashMap is non synchronized.
- Java HashMap maintains no order.

Methods of HashMap

V put(K key, V value)

V get(Object key)

boolean containsKey(Object key)

boolean containsValue(Object value)

V remove(Object key)

void clear()

boolean isEmpty()

int size()

Set<K> keySet()

Collection<V> values()

Set<Map.Entry<K, V>> entrySet()

void putAll(Map<? extends K, ? extends V> m)

V getOrDefault(Object key, V defaultValue):

```
import java.util.*;

public class HashMapExample{

 public static void main(String args[]){

   HashMap<Integer,String> map=new HashMap<Integer,String>();//Creating HashMap

   map.put(1,"Mango");

   map.put(2,"Apple");

   map.put(3,"Banana");

   map.put(4,"Grapes");



   System.out.println("Iterating Hashmap...");

   for(Map.Entry m : map.entrySet()){

    System.out.println(m.getKey()+" "+m.getValue());

   }

}

}
```

Hashtable

- Java Hashtable class contains unique elements.
- Java Hashtable class doesn't allow null key or value.
- Java Hashtable class is synchronized.

Methods of Hashtable:

contains(Object value)
containsKey(Object key)
containsValue(Object value)
elements()
entrySet()
equals(Object o)
get(Object key)
keys()
keySet()

merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction)

put(K key, V value)
putAll(Map<? extends K,? extends V> t)
remove(Object key)
values()
remove(Object key, Object value)

replace(K key, V value)
replace(K key, V oldValue, V newValue)

LinkedHashMap

- Java LinkedHashMap contains values based on the key.
- Java LinkedHashMap contains unique elements.
- Java LinkedHashMap may have one null key and multiple null values.
- Java LinkedHashMap is non synchronized.
- Java LinkedHashMap maintains insertion order.

Method	Description
V get(Object key)	It returns the value to which the specified key is mapped.
void clear()	It removes all the key-value pairs from a map.
boolean containsValue(Object value)	It returns true if the map maps one or more keys to the specified value.
Set<Map.Entry<K,V>> entrySet()	It returns a Set view of the mappings contained in the map.
void forEach(BiConsumer<? super K,? super V> action)	It performs the given action for each entry in the map until all entries have been processed or the action throws an exception.
V getOrDefault(Object key, V defaultValue)	It returns the value to which the specified key is mapped or defaultValue if this map contains no mapping for the key.
Set<K> keySet()	It returns a Set view of the keys contained in the map
protected boolean removeEldestEntry(Map.Entry<K,V> eldest)	It returns true on removing its eldest entry.
void replaceAll(BiFunction<? super K,? super V,? extends V> function)	It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.
Collection<V> values()	It returns a Collection view of the values contained in this map.

```
import java.util.*;
class LinkedHashMap1{
 public static void main(String args[]){
  LinkedHashMap<Integer,String> hm=new LinkedHashMap<Integer,String>();
  hm.put(100,"Amit");
  hm.put(101,"Vijay");
  hm.put(102,"Rahul");
for(Map.Entry m:hm.entrySet()){
   System.out.println(m.getKey()+" "+m.getValue());
  }
 }
}
```

Exception Handling

Exception Handling is the mechanism to handle runtime malfunctions. We need to handle such exceptions to prevent abrupt termination of program. The term exception means exceptional condition, it is a problem that may arise during the execution of program. A bunch of things can lead to exceptions, including programmer error, hardware failures, files that need to be opened cannot be found, resource exhaustion etc.

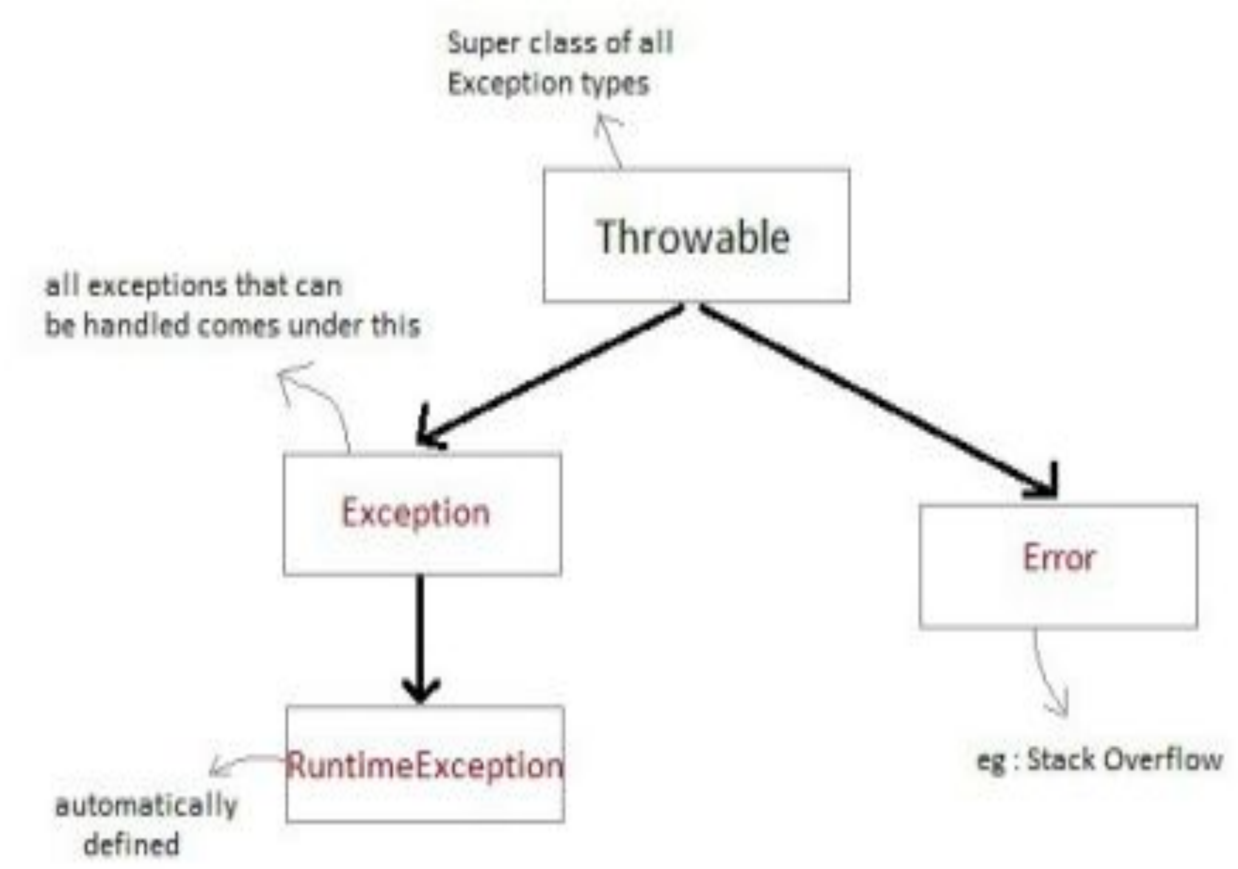
Exception:

Exceptions are conditions or events that alter the normal flow of a program. They are typically used to handle predictable problems that a program might encounter.

Error:

Errors generally refer to more severe issues that occur at runtime or compile-time and are usually outside the control of the program. They indicate problems that the program can't easily recover from.

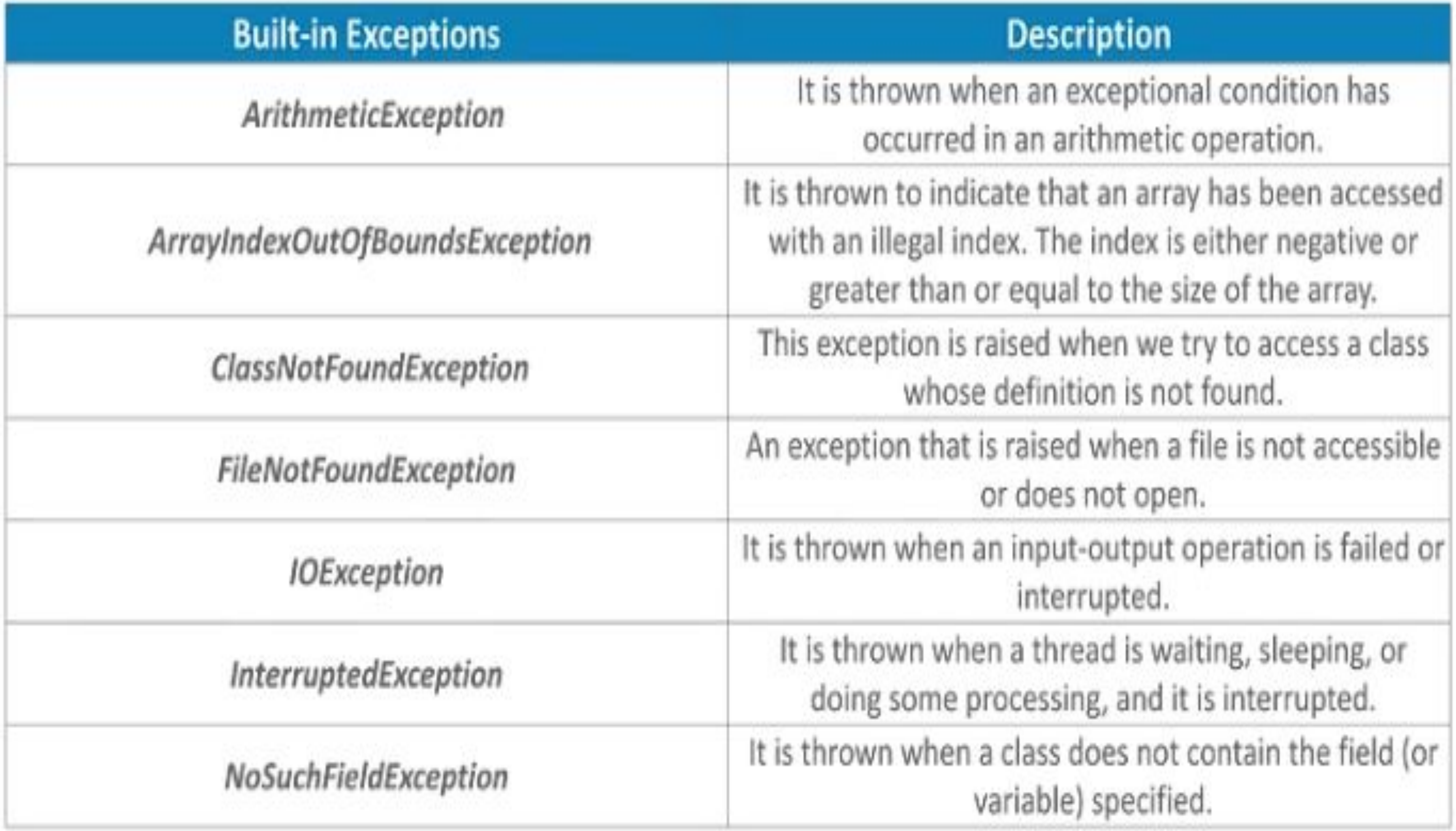
Exception class Hierarchy


Super class of all Exception types
Throwable
all exceptions that can be handled comes under this
Exception
Error
RuntimeException
automatically defined
eg : Stack Overflow

Types of Exception

Exception are categorized into 3 category.

1. **Checked Exception** The exception that can be predicted by the programmer.Example : File that need to be opened is not found. These type of exceptions must be checked at compile time.
2. **Unchecked Exception** Unchecked exceptions are the class that extends RuntimeException. Unchecked exception are ignored at compile time. Example : ArithmeticException, NullPointerException, Array Index out of Bound exception. Unchecked exceptions are checked at runtime.
3. **Error** Errors are typically ignored in code because you can rarely do anything about an error. Example : if stack overflow occurs, an error will arise. This type of error is not possible handle in code.



Built-in Exceptions	Description
ArithmeticException	It is thrown when an exceptional condition has occurred in an arithmetic operation.
ArrayIndexOutOfBoundsException	It is thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.
ClassNotFoundException	This exception is raised when we try to access a class whose definition is not found.
FileNotFoundException	An exception that is raised when a file is not accessible or does not open.
IOException	It is thrown when an input-output operation is failed or interrupted.
InterruptedException	It is thrown when a thread is waiting, sleeping, or doing some processing, and it is interrupted.
NoSuchFieldException	It is thrown when a class does not contain the field (or variable) specified.

Exception Handling Mechanism

In java, exception handling is done using five keywords

1. try
2. catch
3. Throw
4. Throws
5. finally

try….catch

- Try is used to guard a block of code in which exception may occur.
- This block of code is called guarded region.
- A catch statement involves declaring the type of exception you are trying to catch.
- If an exception occurs in guarded code, the catch block that follows the try is checked, if the type of exception that occured is listed in the catch block then the exception is handed over to the catch block which then handles it.

```
class Excp
 {
public static void main(String args[]) {
int a,b,c;
try {
a=0;
b=10;
c=b/a;
System.out.println("This line will not be executed"); } catch(ArithmeticException e)
 { System.out.println("Divided by zero"); }
System.out.println("After exception is handled");
} }
```

Multiple catch blocks:

A try block can be followed by multiple catch blocks. You can have any number of catch blocks after a single try block.If an exception occurs in the guarded code the exception is passed to the first catch block in the list. If the exception type of exception, matches with the first catch block it gets caught, if not the exception is passed down to the next catch block. This continue until the exception is caught or falls through all catches.

```
class Excep

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

 try {

int arr[]={1,2};

 arr[2]=3/0;

} catch(ArithmeticException ae)

{ System.out.println("divide by zero");

} catch(ArrayIndexOutOfBoundsException e)

 { System.out.println("array index out of bound exception");

} } }
```

Nested try statement

- try statement can be nested inside another block of try.
- Nested try block is used when a part of a block may cause one error while entire block may cause another error.
- In case if inner try block does not have a catch handler for a particular exception then the outer try is checked for match.

```
class Excep
 { public static void main(String[] args) {
 try {
 int arr[]={5,0,1,2};
try {
 int x=arr[3]/arr[1];
} catch(ArithmeticException ae)
{ System.out.println("divide by zero");
} arr[4]=3;
} catch(ArrayIndexOutOfBoundsException e)
{ System.out.println("array index out of bound exception");
} } }
```

finally

- In Java, the finally keyword is used with try and catch blocks to ensure that a block of code is executed regardless of whether an exception is thrown or not.
- It is commonly used for code that needs to run after a try block, such as closing resources (files, network connections, etc.) or performing cleanup operations.

```
class Excp
 {
public static void main(String args[]) {
int a,b,c;
try {
a=0;
b=10;
c=b/a;
System.out.println("This line will not be executed"); } catch(ArithmeticException e)
 { System.out.println("Divided by zero"); }
System.out.println("After exception is handled");
finally{System.out.println("Code is always executed”);}
}
}
```

throws

- The throws keyword is used for checked exceptions, which are exceptions that the Java compiler requires to be either caught or declared in the method signature. Checked exceptions are subclasses of Exception but not of RuntimeException.
- The throws clause is part of the method's signature. This means that if you override a method, you must declare that it throws the same exceptions or subclasses of the exceptions declared in the parent method's throws clause.

Creating Instance of Throwable class

There are two possible ways to get an instance of class Throwable,

1. Using a parameter in catch block.

2. Creating instance with new operator.

3. new NullPointerException("test");

This constructs an instance of NullPointerException with name test.

```
class Test{
static void check() throws ArithmeticException{
System.out.println("Inside check function");
throw new ArithmeticException("demo");}
public static void main(String args[]){
try{
check();}
catch(ArithmeticException e){
System.out.println("caught" + e);
}}}
```

throw Keyword

- throw keyword is used to throw an exception explicitly. Only object of Throwable class or its sub classes can be thrown.
- Program execution stops on encountering throw statement, and the

  closest catch statement is checked for matching type of exception.
- Syntax :

  type method_name(parameter_list)throwsexception_list

```
class Test{

static void avg(){

try{

throw new ArithmeticException("demo exception");}

catch(ArithmeticException e){

System.out.println("Exception caught");}}

public static void main(String args[]){

avg();

}

}
```

User defined Exception

Steps for creating user defined exception:

1. Extend the Exception class to create your own **Exception** class.
2. You don't have to implement anything inside it, no methods are required.
3. You can have a Constructor if you want.
4. You can override the **toString()** function, to display customized message.

```
class VotingException extends Exception {

    public VotingException (String message) {
        super(message);
    }
}
```

```
public class VotingEligibilityChecker {

    public void checkAge(int age) throws VotingException {
        if (age < 18) {
            throw new VotingException ("Not eligible for voting. Age must be 18 or older.");
        } else {
            System.out.println("Eligible for voting.");
        }
    }
public static void main(String[] args) {
      VotingEligibilityChecker checker = new VotingEligibilityChecker();
      int age = 16;
      try {
          checker.checkAge(age);
      } catch( VotingException e) {
          System.out.println(e.getMessage());
      }
   }
}
```

String class

String class is part of the **java.lang** package and is used to represent a sequence of characters.

Strings in Java are **immutable**, which means once a String object is created, it cannot be changed. Instead, operations on strings create new String objects.

Methods

1. int length()
2. boolean isEmpty()
3. char charAt(int index)
4. boolean contains(CharSequence sequence)
5. boolean equals(Object anObject)
6. boolean equalsIgnoreCase(String anotherString)
7. int indexOf(String str)
8. int lastIndexOf(String str)
9. String substring(int beginIndex)
10. String substring(int beginIndex, int endIndex)
11. String trim()
12. String toLowerCase()
13. String toUpperCase()

StringBuffer

- The **StringBuffer** class in Java is part of the java.lang package
- It is used to create **mutable** sequences of characters. Unlike String, which is immutable.
- It is **Synchronized**.

Methods:

1. StringBuffer append(String str)
2. StringBuffer insert(int offset, String str)
3. StringBuffer delete(int start, int end)
4. StringBuffer replace(int start, int end, String str)
5. StringBuffer reverse()
6. String toString()
7. int length()

```
public class StringBufferDemo {

    public static void main(String[] args) {

        StringBuffer sb = new StringBuffer("Hello");

        sb.append(" World");

        sb.append('!');

        System.out.println("After append: " + sb);

        sb.insert(6, " Java");

        System.out.println("After insert: " + sb);

        sb.replace(6, 10, "Programming");

        System.out.println("After replace: " + sb);

        sb.delete(6, 17);
```

```
 System.out.println("After delete: " + sb);

        sb.reverse();

        System.out.println("After reverse: " + sb);

        System.out.println("Length: " + sb.length());

        String str = sb.toString();

        System.out.println("Converted to String: " + str);

    }

}
```

File Class

File class is part of the java.io package and provides an abstraction for files and directories.

It allows you to work with files and directories on your filesystem, including creating, deleting, and querying them.

```
File file = new File("C:\\path\\to\\your\\file.txt");

File relativeFile = new File("file.txt");
```

Methods

createNewFile()

canWrite()

canExecute()

canRead()

isFile()

getName()

listFiles()

```
import java. Io *;

class ReadTest{

public static void main(String[] args){

try{

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

BufferedReader br = new BufferedReader(new FileReader(fl)) ;

String str;

while ((str=br.readLine())!=null){

System.out.println(str);}

br.close();

fl.close();}

catch (IOException e)

{ e.printStackTrace(); }}

}
```

```
import java. Io *;

class WriteTest{

public static void main(String[] args){

try{

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

String str="Write this string to my file";

FileWriter fw = new FileWriter(fl) ;

fw.write(str);

fw.close();

fl.close();}

catch (IOException e)

{ e.printStackTrace(); }}}
```

DataInputStream Class

Reads primitive data types from an input stream.

Methods:

- readInt(): Reads an int value.
- readFloat(): Reads a float value.
- readChar(): Reads a char value.
- readBoolean(): Reads a boolean value.
- readUTF(): Reads a String in UTF format.

```
public class DataInputStreamExample {
    public static void main(String[] args) {
        try {            FileInputStream fis = new FileInputStream("datafile.txt");
            DataInputStream dis = new DataInputStream(fis);
            int num = dis.readInt();
            float decimal = dis.readFloat();
            String text = dis.readUTF();
            System.out.println("Number: " + num);
            System.out.println("Decimal: " + decimal);
            System.out.println("Text: " + text);
            dis.close();
        } catch (IOException e) {            e.printStackTrace();        }    }}
```

DataOutputStream Class

Writes primitive data types to an output stream.

Methods:

- writeInt(int v): Writes an int value.
- writeFloat(float v): Writes a float value.
- writeChar(int v): Writes a char value.
- writeBoolean(boolean v): Writes a boolean value.
- writeUTF(String s): Writes a String in UTF format.

```
public class DataOutputStreamExample {
  public static void main(String[] args) {        try {
          FileOutputStream fos = new FileOutputStream("datafile.txt");
          DataOutputStream dos = new DataOutputStream(fos);
          dos.writeInt(123);
          dos.writeFloat(45.67f);
          dos.writeUTF("Hello, World!");
          dos.close();
      } catch (IOException e) {
          e.printStackTrace();
      }
  }}
```