String, StringBuffer, and StringBuilder

1. String

- **Definition**: Immutable sequence of characters.
- **Immutability**: Once created, the value of a String object cannot be modified. Any operation that changes the string creates a new object.
- **Thread-Safety**: Not thread-safe.
- **Use Case**: Suitable for storing and working with constant data.

Common Methods in the String Class:

Method	Description	Example
char charAt(int index)	Returns the character at the specified index.	str.charAt(1);
int length()	Returns the length of the string.	str.length();
String substring(int beginIndex)	Returns a substring starting from the given index.	str.substring(6);
String substring(int beginIndex, int endIndex)	Returns a substring between start (inclusive) and end (exclusive).	str.substring(0, 5);
boolean contains(CharSequence s)	Checks if the string contains the specified character sequence.	str.contains("Java");
boolean isEmpty()	Checks if the string is empty.	str.isEmpty();
String concat(String str)	Concatenates the specified string.	str1.concat(str2);
String replace(char old, char new)	Replaces all occurrences of the specified character.	str.replace('a', 'o');
boolean equalsIgnoreCase(String another)	Compares strings, ignoring case.	str1.equalsIgnoreCase(str2);
String[] split(String regex)	Splits the string based on a regular expression.	str.split(",");
int indexOf(int ch)	Returns the index of the first occurrence of the specified character.	str.indexOf('l');

Method	Description	Example
String toLowerCase()	Converts the string to lowercase.	str.toLowerCase();
String toUpperCase()	Converts the string to uppercase.	str.toUpperCase();
String trim()	Removes leading and trailing spaces.	str.trim();
static String valueOf(int value)	Converts a given value to a string.	String.valueOf(123);

---

2. StringBuffer

- **Definition**: Mutable sequence of characters.
- **Mutability**: Allows modification of characters or content without creating a new object.
- **Thread-Safety**: Thread-safe because methods are synchronized.
- **Use Case**: Suitable for multi-threaded applications where string modifications are required.

Common Methods in the StringBuffer Class:

Method	Description	Example
append(String str)	Appends the specified string.	sb.append("Java");
insert(int offset, String str)	Inserts a string at the specified position.	sb.insert(4, "Lang");
replace(int start, int end, String str)	Replaces characters in a specified range.	sb.replace(0, 4, "Hi");
delete(int start, int end)	Deletes characters in a specified range.	sb.delete(0, 4);
reverse()	Reverses the string.	sb.reverse();
capacity()	Returns the current capacity.	sb.capacity();
charAt(int index)	Returns the character at the specified index.	sb.charAt(1);

---

3. StringBuilder

- **Definition**: Mutable sequence of characters.
- **Mutability**: Similar to StringBuffer, but more efficient for single-threaded applications.
- **Thread-Safety**: Not thread-safe because methods are not synchronized.

- **Use Case**: Suitable for single-threaded applications where string modifications are required.

Common Methods in the StringBuilder Class:

Method	Description	Example
append(String str)	Appends the specified string.	sb.append("Java");
insert(int offset, String str)	Inserts a string at the specified position.	sb.insert(4, "Lang");
replace(int start, int end, String str)	Replaces characters in a specified range.	sb.replace(0, 4, "Hi");
delete(int start, int end)	Deletes characters in a specified range.	sb.delete(0, 4);
reverse()	Reverses the string.	sb.reverse();
capacity()	Returns the current capacity.	sb.capacity();
charAt(int index)	Returns the character at the specified index.	sb.charAt(1);

Key Differences Between String, StringBuffer, and StringBuilder

Feature	String	StringBuffer	StringBuilder
Mutability	Immutable	Mutable	Mutable
Thread-Safety	Not thread-safe	Thread-safe (synchronized)	Not thread-safe
Performance	Slower in modifications	Slower due to synchronization	Faster for single-threaded operations
Use Case	Constant data or fewer modifications	Multi-threaded environments	Single-threaded environments

---

Examples

1. Using String

```
String str = "Hello";

str = str.concat(" World");

System.out.println(str); // Output: "Hello World"
```

2. Using StringBuffer

```
StringBuffer sb = new StringBuffer("Hello");
```

```
sb.append(" World");
System.out.println(sb); // Output: "Hello World"
```

3. Using StringBuilder

```
StringBuilder sb = new StringBuilder("Hello");
sb.append(" World");
System.out.println(sb); // Output: "Hello World"
```

What is **garbage collection**?

how does garbage collection works?

Does garbage collection happens automatically ?

can we call garbage collector explicitly?

what is the use of finalize method?

Data Stored in the Heap Area in Java

In Java, the **heap memory** is the runtime memory where objects and their instance variables (non-static fields) are stored. The heap is managed by the **Garbage Collector (GC)**.

Types of Data Stored in the Heap:

1. **Objects**
    - All objects created using new keyword are stored in the heap.
    - Example:

```
class Person {
  String name;
}


public class Test {
  public static void main(String[] args) {
    Person p1 = new Person(); // Object stored in heap
  }
}
```

2. **Instance Variables (Non-static Fields)**
    - Variables that belong to an object (instance variables) are stored within the object in the heap.
    - Example:

```
class Employee {
  int id;      // Stored in heap (inside the object)
  String name;   // Stored in heap (inside the object)
}
```

3. **String Objects (if created using new)**
    - If a String is created using new, it is stored in the **heap**.
    - Example:

String str1 = new String("Hello"); // Stored in heap

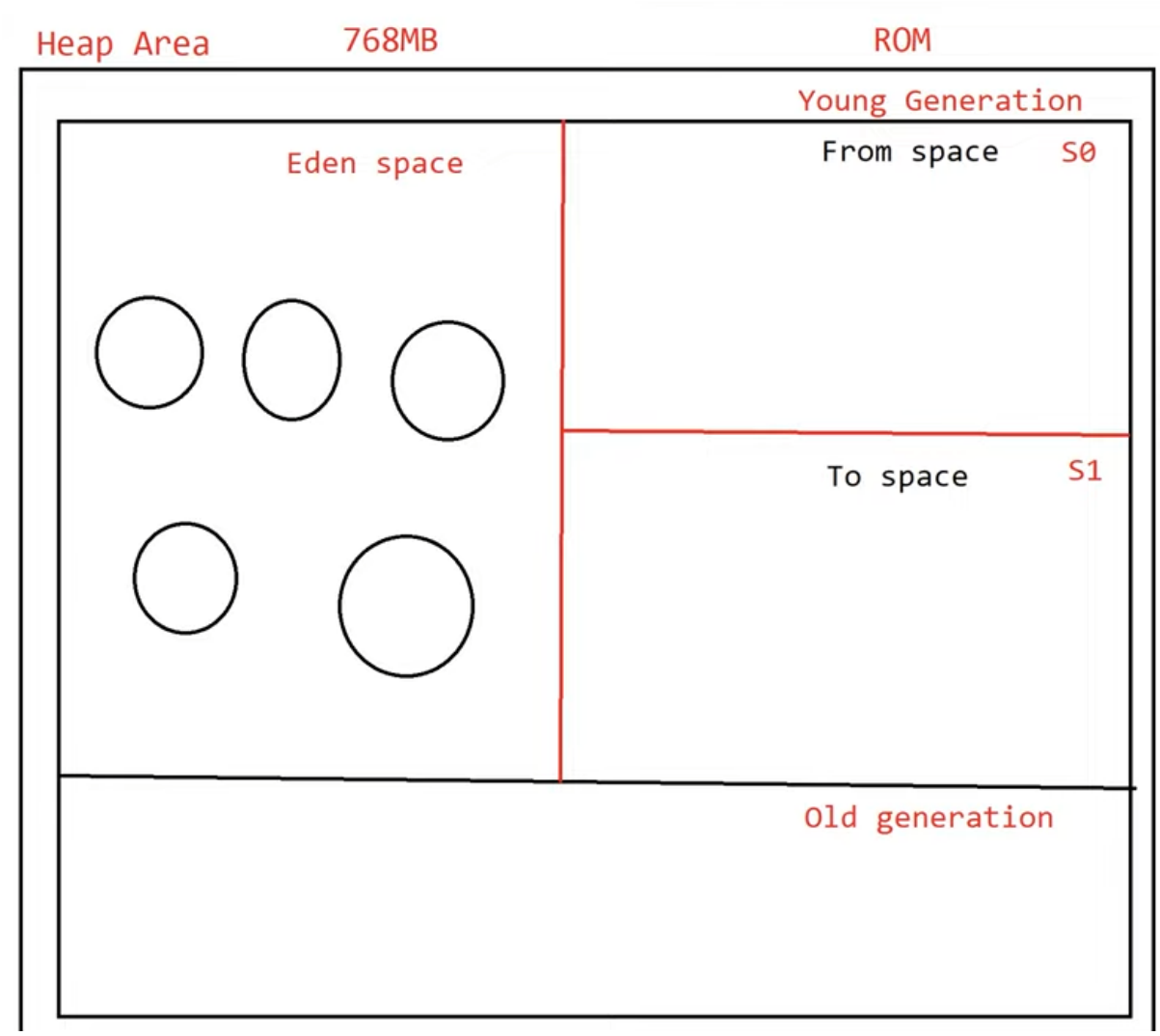
- However, **String literals** are stored in the **String Constant Pool (SCP),** which is a part of the heap.

4. **Arrays (of Objects or Primitives)**
    - Arrays in Java are objects, so they are stored in the heap.
    - Example:

int[] arr = new int[5]; // Array object stored in heap

What is Not Stored in the Heap?

- **Method Area (inside Heap) stores:**
    - Class metadata (class names, method information, etc.)
    - Static variables
    - Constant pool
- **Stack Memory stores:**
    - Local variables
    - Method call information (stack frames)



Garbage Collection (gc()) and finalize() in Java

---

1. gc() Method (Garbage Collection)

What is gc()?

- gc() is a method in System and Runtime classes that **requests** the Java Virtual Machine (JVM) to run the garbage collector.
- It **does not guarantee** immediate garbage collection; it is just a **request** to the JVM.

Syntax:

System.gc(); // Recommended way to request GC

or

Runtime.getRuntime().gc(); // Alternative way to request GC

Example:

```
public class GCDemo {
  public static void main(String[] args) {
    GCDemo obj1 = new GCDemo();
    obj1 = null;  // Making object eligible for garbage collection

    System.gc();  // Requesting JVM to run GC

    System.out.println("End of main method.");
  }

  @Override
  protected void finalize() {
    System.out.println("Garbage collector called!");
  }
}
```

Key Points About gc():

- **It is non-deterministic** (JVM may or may not run GC immediately).
- **Garbage Collector runs in the background** and automatically manages memory.
- **Explicit calls to gc() are discouraged** as JVM optimizes memory management itself.
- **Garbage collection occurs in heap memory** and removes unreferenced objects.

---

2. finalize() Method

What is finalize()?

- finalize() is a **protected method** of the Object class, which is called by the garbage collector **before** an object is destroyed.
- It was used to **release resources** before an object is garbage collected.
- **Deprecated in Java 9** and **removed in Java 18** because it was unreliable.

Syntax:

```
@Override
protected void finalize() throws Throwable {
```

```
    // Cleanup code (not recommended)
}
```

Example:

```
class FinalizeDemo {
    @Override
    protected void finalize() throws Throwable {
        System.out.println("finalize() method called");
    }

    public static void main(String[] args) {
        FinalizeDemo obj = new FinalizeDemo();
        obj = null;  // Object becomes eligible for GC
         new FinalizeDemo();
        System.gc();  // Requesting GC
        System.out.println("Main method execution completed.");
    }
}
```

Key Points About finalize():

- Used for **resource cleanup**, but **not recommended** (better alternatives: try-with-resources, finally block).
- JVM **does not guarantee** finalize() will be called before GC.
- **Slow and unreliable**, leading to its deprecation in Java 9 and removal in Java 18.

INNER CLASSES

Types of Inner Classes in Java

Java supports different types of **inner classes**, which are classes defined inside another class. The main types are:

1. **Normal Inner Class (Member Inner Class)**
2. **Local Inner Class**
3. **Anonymous Inner Class**
4. **Static Nested Class**

---

1. Normal Inner Class (Member Inner Class)

- A **non-static class inside another class**.
- It has access to the **outer class's members (including private members)**.
- Requires an **instance of the outer class** to be created.

Example:

```
class Outer {
  class Inner {  // Normal Inner Class
    void display() {
      System.out.println("Inside Normal Inner Class");
    }
  }

  public static void main(String[] args) {
    Outer outer = new Outer();
    Outer.Inner inner = outer.new Inner(); // Creating inner class object
    inner.display();
  }
}
```

Key Points:

- Can **access all members of the outer class**.
- Requires an **instance of the outer class** to create an inner class object.
- Cannot define **static members** inside (except static final constants).

---

2. Local Inner Class

- A class **defined inside a method or block**.
- **Scope is limited** to the method/block where it is defined.
- Can access **final or effectively final** variables from the enclosing method.

Example:

```
class Outer {
  void outerMethod() {
    class LocalInner {  // Local Inner Class
      void show() {
        System.out.println("Inside Local Inner Class");
      }
    }
    LocalInner local = new LocalInner();
    local.show();
  }

  public static void main(String[] args) {
    new Outer().outerMethod();
  }
}
```

Key Points:

- Cannot have **static members**.
- Exists **only while the method is executing**.
- Can access **final or effectively final variables** from the enclosing method.

---

3. Anonymous Inner Class

- A class **without a name** that extends a class or implements an interface.
- **Defined inside a method or block**.
- Used for **short-lived implementations**.

Example: Using Anonymous Class with Interface

```
interface Greeting {
   void sayHello();
}
public class Test {
   public static void main(String[] args) {
      Greeting g = new Greeting() { // Anonymous Inner Class
         public void sayHello() {
            System.out.println("Hello from Anonymous Inner Class!");
         }
      };
      g.sayHello();
   }
}
```

Example: Using Anonymous Class with Class

```
abstract class Animal {
   abstract void sound();
}

public class Test {
   public static void main(String[] args) {
      Animal cat = new Animal() {  // Anonymous Inner Class
         void sound() {
            System.out.println("Meow!");
         }
      };
      cat.sound();
   }
}
```

Key Points:

- Cannot have **explicit constructors**.
- Used for **quick, one-time use cases**.
- Cannot have **multiple methods (except for overriding inherited ones)**.

---

4. Static Nested Class

- A **static class inside another class**.
- **Can have static and non-static members**.
- **Does not require an outer class instance** to create an object.

Example:

```
class Outer {
   static class StaticNested {  // Static Nested Class
      void display() {
         System.out.println("Inside Static Nested Class");
      }
   }

   public static void main(String[] args) {
      Outer.StaticNested nested = new Outer.StaticNested();  // No outer instance needed
      nested.display();
   }
}
```

Key Points:

- Can access **only static members** of the outer class.
- Can have **static and non-static members**.
- Does **not require an instance of the outer class**.

EXCEPTION HANDLING

Error vs Exception in Java

1. **Error:**
    - An **Error** in Java represents a serious problem that a program cannot typically recover from.
    - It is usually caused by the JVM (Java Virtual Machine) or the system itself.
    - Errors are not meant to be caught by the application; they generally indicate a failure in the JVM or the environment.
    - **Examples of errors**:
        - OutOfMemoryError: When the JVM runs out of memory.
        - StackOverflowError: When the stack overflows due to deep recursion.

Errors are unchecked (i.e., they are not required to be caught or declared).

2. **Exception:**
    - An **Exception** is a problem that arises during the execution of the program. It can often be caught and handled by the program.
    - Exceptions can be further divided into **checked exceptions** and **unchecked exceptions**.

Types of exceptions:

- **Checked exceptions**: Exceptions that are checked at compile-time, and must either be caught or declared in the method signature using throws.
    - Examples:
        - IOException

        - SQLException
    - These exceptions are typically recoverable and can be handled by the programmer.
- **Unchecked exceptions**: These extend RuntimeException, and they are not checked at compile-time.
    - Examples:
        - NullPointerException
        - ArrayIndexOutOfBoundsException
        - ArithmeticException
    - Unchecked exceptions are often caused by programming errors or bugs. They usually do not need to be explicitly handled.

What is Exception Handling?

Exception handling in Java is a mechanism that allows you to manage runtime errors, ensuring the normal flow of the program is not disrupted. It allows you to catch and respond to exceptions, giving your program the ability to recover or terminate gracefully.

Java provides a structured approach to exception handling using the following keywords:

1. **try**: Block of code where exceptions might occur.
2. **catch**: Block of code to handle exceptions.
3. **finally**: A block that runs whether an exception occurs or not (used for cleanup code like closing files or database connections).
4. **throw**: Used to explicitly throw an exception.
5. **throws**: Declares that a method may throw an exception.

Example of exception handling:

```
try {
   int result = 10 / 0;  // Division by zero will cause ArithmeticException
} catch (ArithmeticException e) {
   System.out.println("Caught an exception: " + e.getMessage());
} finally {
   System.out.println("This will always execute, regardless of exception.");
}
```

Types of Exceptions We Handle in Java

Java provides two main categories of exceptions that can be handled:

1. **Checked Exceptions**:

   Checked exceptions in Java are exceptions that the compiler forces you to either **handle** or **declare** in your code.

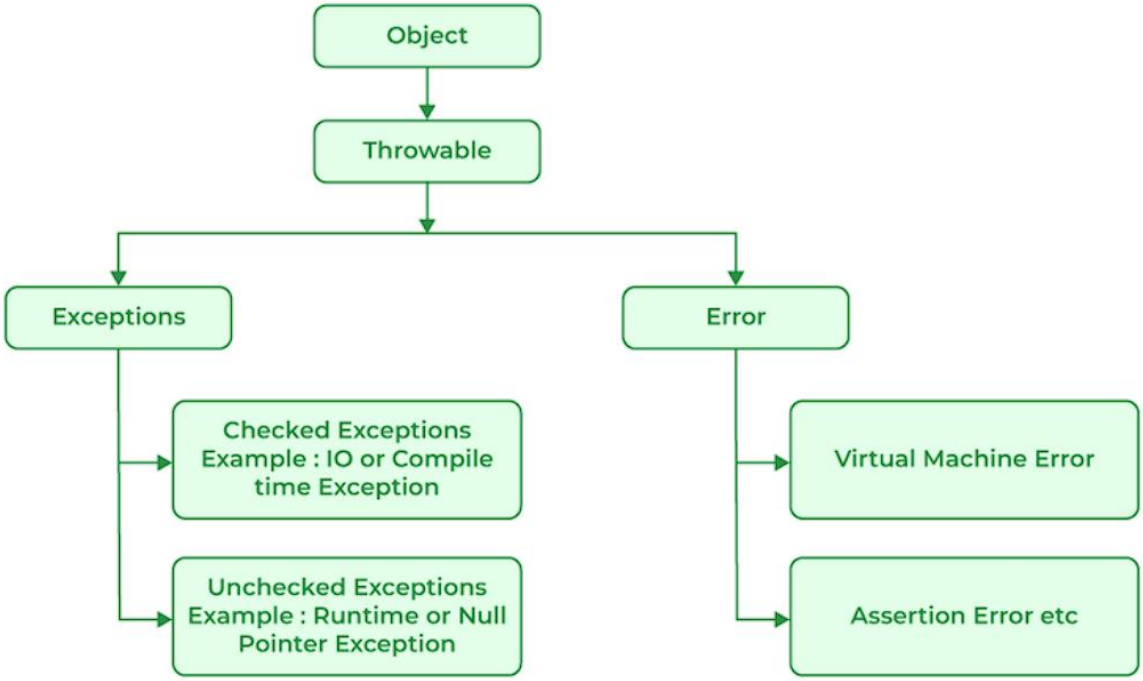
   In simple terms:

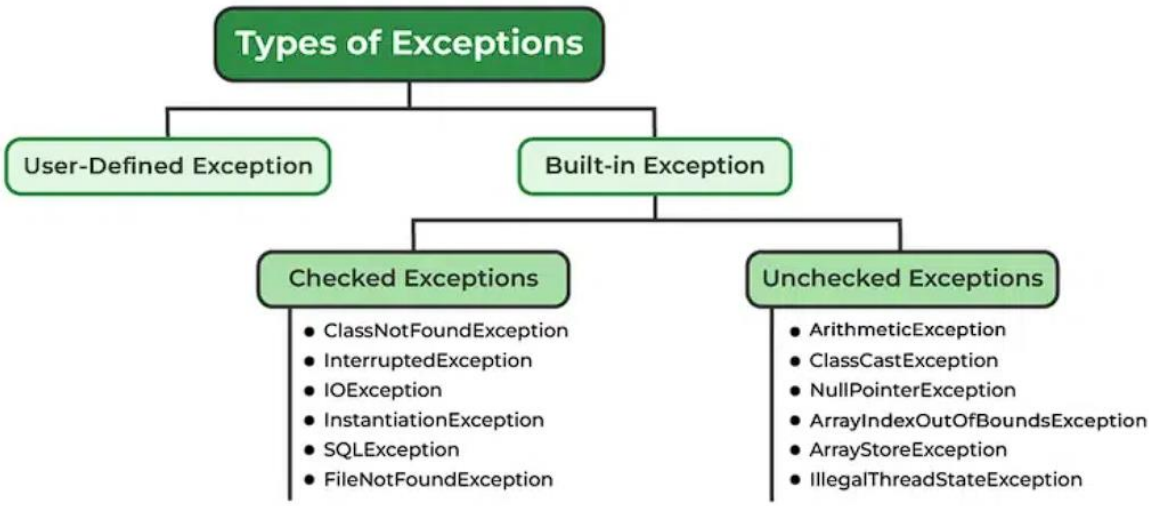
- **Checked exceptions** are exceptions that the compiler **knows about** and expects you to deal with.
- They usually occur due to external factors, like file operations, database issues, or network problems, which you can predict and try to handle in advance.
- If you don't handle a checked exception, the compiler will give you an error.

  To handle a checked exception, you must either:

  1. Use a try-catch block, or
  2. Declare the exception with the throws keyword in the method signature

2. **Unchecked Exceptions (Runtime Exceptions)**:
   - These exceptions are not checked at compile-time, but they occur during runtime.
   - These exceptions are typically caused by programming errors.
   - **Examples**:
     - NullPointerException: Trying to call a method on a null object.
     - ArrayIndexOutOfBoundsException: Trying to access an index outside the bounds of an array.
     - ArithmeticException: Performing an invalid arithmetic operation, like dividing by zero.


Object
Throwable
Exceptions
Error
Checked Exceptions Example : IO or Compile time Exception
Unchecked Exceptions Example : Runtime or Null Pointer Exception
Virtual Machine Error
Assertion Error etc


Types of Exceptions
User-Defined Exception
Built-in Exception
Checked Exceptions
ClassNotFoundException
InterruptedException
IOException
InstantiationException
SQLException
FileNotFoundException
Unchecked Exceptions
ArithmeticException
ClassCastException
NullPointerException
ArrayIndexOutOfBoundsException
ArrayStoreException
IllegalThreadStateException

PriorityQueue Methods:

The PriorityQueue inherits methods from the Queue and Collection interfaces. Here are the commonly used methods:

1. Adding Elements

Method	Description
add(E e)	Inserts an element. Throws IllegalStateException if capacity is full.
offer(E e)	Inserts an element. Returns false if the queue is full (instead of an exception).

2. Removing Elements

Method	Description
remove()	Removes and returns the head. Throws NoSuchElementException if empty.
poll()	Removes and returns the head. Returns null if empty.
clear()	Removes all elements from the queue.

3. Retrieving (Accessing) Elements

Method	Description
peek()	Retrieves (but does not remove) the head. Returns null if empty.
element()	Retrieves (but does not remove) the head. Throws NoSuchElementException if empty.

4. Checking Properties

Method	Description
size()	Returns the number of elements in the queue.
isEmpty()	Returns true if the queue is empty.
contains(Object o)	Checks if an element is present.

5.Converting to Array

Method	Description
toArray()	Returns an Object[] containing all elements.

Set Interface:

The Set interface in Java is a part of the java.util package and extends the Collection interface. It represents a collection of unique elements (no duplicates). Some commonly used classes that implement the Set interface are HashSet, LinkedHashSet, and TreeSet.

Methods in Set Interface

Since Set extends Collection, it inherits all the methods from Collection. Below are the commonly used methods:

Basic Methods

1. **add(E e)** – Adds an element to the set if it's not already present.
2. **addAll(Collection<? extends E> c)** – Adds all elements from the specified collection to this set.
3. **clear()** – Removes all elements from the set.
4. **contains(Object o)** – Returns true if the set contains the specified element.
5. **containsAll(Collection<?> c)** – Returns true if the set contains all elements of the specified collection.
6. **isEmpty()** – Returns true if the set is empty.
7. **iterator()** – Returns an iterator for iterating over the elements in the set.
8. **remove(Object o)** – Removes the specified element from the set if present.
9. **removeAll(Collection<?> c)** – Removes all elements from the set that exist in the specified collection.
10. **retainAll(Collection<?> c)** – Retains only the elements in this set that are contained in the specified collection.
11. **size()** – Returns the number of elements in the set.
12. **toArray()** – Returns an array containing all elements of the set.

Key Differences Between HashSet and LinkedHashSet

Feature	**HashSet**	**LinkedHashSet**
Ordering	No specific order	Maintains insertion order
Performance	Slightly faster	Slightly slower due to linked list
Memory Usage	Less	More (because of the doubly linked list)
Best Use Case	When order does not matter	When insertion order matters

TreeSet Methods in Java:

`1. Navigating Elements (NavigableSet Methods)

- lower(E e) → Returns the greatest element less than the given element.
- floor(E e) → Returns the greatest element ≤ given element.
- higher(E e) → Returns the smallest element greater than the given element.
- ceiling(E e) → Returns the smallest element ≥ given elemen

2. HeadSet, TailSet, and SubSet

- headSet(E e) → Returns elements less than the given element.
- tailSet(E e) → Returns elements greater than or equal to the given element.
- subSet(E from, E to) → Returns elements in a range [from, to).

3. Descending Order

- **descendingSet()** → Returns a reverse-ordered view of the set.

Key Differences Between HashSet, LinkedHashSet, and TreeSet

Feature	HashSet	LinkedHashSet	TreeSet
Ordering	No specific order	Maintains insertion order	Sorted (ascending by default)
Performance	Fastest (O(1))	Slightly slower	Slower (O(log n))
Duplicate Elements	Not allowed	Not allowed	Not allowed
Null Allowed?	Yes (only one)	Yes (only one)	No (Throws NullPointerException)
Best Use Case	Fast lookups	Maintaining insertion order	Sorted unique elements