**Comprehensive List of Useful Methods in Java for Problem Solving**

This list combines methods from various data structures, the Math library, and other helpful functionalities:

Data Structures:

* Arrays: A collection of elements of the same data type, stored in contiguous memory locations and accessed by an index.
* Strings: A sequence of characters representing text.
* LinkedLists: A linear data structure where elements are not stored in contiguous memory locations, but each element contains a reference to the next element in the sequence.
* ArrayList: A resizable-array implementation of the List interface, providing constant-time performance for basic operations like getting and setting elements.
* Stacks: A linear data structure that follows the Last-In-First-Out (LIFO) principle, where the last element added is the first one to be removed.
* Queues: A linear data structure that follows the First-In-First-Out (FIFO) principle, where the first element added is the first one to be removed.
* Sets: A collection of unique elements, where each element appears only once.
  + HashSet: An unordered set implementation that uses a hash table for storage, providing constant-time performance for basic operations like adding and checking for elements.
  + TreeSet: An ordered set implementation that stores elements in a red-black tree, maintaining them in sorted order.
* Maps: A data structure that stores key-value pairs, where each key is unique.
  + HashMap: An unordered map implementation that uses a hash table for storage, providing constant-time performance for basic operations like adding, retrieving, and removing key-value pairs.
  + TreeMap: An ordered map implementation that stores key-value pairs in a red-black tree, maintaining them in sorted order based on the keys.
* Trees: A hierarchical data structure where data is organized in nodes, with each node potentially having child nodes.
  + Binary Search Tree: A tree data structure where each node has at most two children, and the values in the left subtree are less than the root, while the values in the right subtree are greater than the root.
* Graphs: A non-linear data structure consisting of nodes (vertices) and edges that connect these nodes, used to represent relationships or connections between objects.

Array Methods:

* length: Returns the number of elements in the array.
* clone(): Creates a shallow copy of the array.
* Arrays.sort(): Sorts the elements of the array in ascending order.
* System.arraycopy(): Copies a range of elements from one array to another.
* indexOf(element, fromIndex): Returns the index of the first occurrence of the specified element in the array, starting from the specified index.
* lastIndexOf(element, fromIndex): Returns the index of the last occurrence of the specified element in the array, starting from the specified index.
* forEach(callback): Executes a provided function once for each element in the array.
* map(callback): Creates a new array with the results of calling a provided function on every element in the original array.
* filter(callback): Creates a new array with all elements that pass the test implemented by the provided function.
* reduce(callback, initialValue): Applies a function against an accumulator and each element in the array, reducing it to a single value.
* push(element): Appends an element to the end of the array and returns the new length.
* pop(): Removes the last element from the array and returns it.
* shift(): Removes the first element from the array and returns it.
* unshift(element1, element2, ...): Inserts new elements at the beginning of the array and returns the new length.
* splice(start, deleteCount, item1, item2, ...): Removes elements from the array from a specified start index and optionally replaces them with provided new items.
* reverse(): Reverses the order of elements in the array in-place (modifies the original array).
* join(separator): Creates and returns a string by concatenating all array elements, joined with the specified separator.
* slice(start, end): Extracts a section of the array and returns a new array.
* fill(value, start, end): Fills all or part of the array with a static value.
* concat(array1, array2, ...): Merges multiple arrays into a single new array.

String Methods:

* length(): Returns the length of the string (the number of characters).
* charAt(index): Returns the character at the specified index in the string.
* substring(startIndex, endIndex): Returns a new string that is a substring of the original string, starting at the specified startIndex and ending at the specified endIndex (not included).
* equals(otherString): Compares two strings for equality and returns a boolean value.
* indexOf(char/String): Returns the index within the string of the first occurrence of the specified character or substring.
* StringBuilder: A mutable sequence of characters that provides an efficient way to build and manipulate strings.
* String.valueOf(): Returns the string representation of an object or primitive data type.
* toCharArray(): Converts the string to a character array.
* concat(str): Returns a new string that is the concatenation of the current string and the specified string str.
* lastIndexOf(char/String): Returns the index within the string of the last occurrence of the specified character or substring.
* startsWith(prefix): Checks if the string starts with the specified prefix and returns a boolean value.
* endsWith(suffix): Checks if the string ends with the specified suffix and returns a boolean value.
* compareTo(otherString): Compares the current string with another string lexicographically and returns an integer value.
* toUpperCase(): Returns a new string with all characters converted to uppercase.
* toLowerCase(): Returns a new string with all characters converted to lowercase.
* trim(): Returns a new string with leading and trailing whitespace characters removed.
* replace(oldChar, newChar): Returns a new string where all occurrences of the specified character are replaced with another character.
* replaceAll(oldString, newString): Returns a new string where all occurrences of the specified substring are replaced with another substring.
* matches(regex): Checks if the string matches the specified regular expression and returns a boolean value.
* replaceFirst(regex, replacement): Replaces the first occurrence of a substring that matches the regular expression with another string.
* replaceAll(regex, replacement): Replaces all occurrences of substrings that match the regular expression with another string.
* split(delimiter): Splits the string into an array of substrings based on the specified delimiter.
* isEmpty(): Checks if the string is empty (length is zero) and returns a boolean value.
* isBlank(): Checks if the string is empty or contains only whitespace characters and returns a boolean value.
* join(delimiter, elements): Joins elements of an array into a single string, separated by the specified delimiter (Java 8 and above).

LinkedList Methods:

* add(element): Adds an element to the end of the list.
* add(index, element): Inserts an element at the specified index in the list.
* get(index): Retrieves the element at the specified index in the list.
* remove(index): Removes the element at the specified index from the list.
* size(): Returns the number of elements in the list.
* isEmpty(): Checks if the list is empty and returns a boolean value.
* set(index, element): Replaces the element at the specified index with a new element.
* indexOf(element): Returns the index of the first occurrence of the specified element in the list.
* lastIndexOf(element): Returns the index of the last occurrence of the specified element in the list.
* clear(): Removes all elements from the list.
* first(): Returns the first element in the list (without removing it).
* last(): Returns the last element in the list (without removing it).
* iterator(): Returns an iterator object to iterate through the elements in the list.
* peek(): Returns the first element in the list without removing it (similar to first()).
* poll(): Removes and returns the first element from the list.
* offer(element): Adds an element to the end of the list (similar to add()).

Stack Methods:

* push(element): Adds an element to the top of the stack.
* pop(): Removes and returns the element from the top of the stack.
* peek(): Returns the element at the top of the stack without removing it.
* isEmpty(): Checks if the stack is empty and returns a boolean value.
* size(): Returns the number of elements in the stack (might not be available in all implementations).
* search(element): Finds the position (distance from the top) of a specific element in the stack (might not be available in all implementations).
* clear(): Removes all elements from the stack (might not be available in all implementations).
* contains(element): Checks if the stack contains a specific element and returns a boolean value (less common, might not be available in all implementations).
* toArray(): Converts the stack to an array (less common, might not be available in all implementations).

Queue Methods:

* offer(element): Adds an element to the back of the queue (enqueue).
* poll(): Removes and returns the element from the front of the queue (dequeue).
* peek(): Returns the element at the front of the queue without removing it.
* isEmpty(): Checks if the queue is empty and returns a boolean value.
* size(): Returns the number of elements in the queue (might not be available in all implementations).
* element(): Returns the element at the front of the queue without removing it (similar to peek, but throws an exception if the queue is empty).
* remove(): Removes and returns the element from the front of the queue (similar to poll, but throws an exception if the queue is empty).
* contains(element): Checks if the queue contains a specific element and returns a boolean value (less common, might not be available in all implementations).
* clear(): Removes all elements from the queue.
* toArray(): Converts the queue to an array (less common, might not be available in all implementations).
* iterator(): Returns an iterator object to iterate through the elements in the queue (less common, might not be available in all implementations).

Set Methods (HashSet):

* add(element): Adds an element to the set.
* contains(element): Checks if the set contains a specific element and returns a boolean value.
* remove(element): Removes an element from the set.
* isEmpty(): Checks if the set is empty and returns a boolean value (might not be available in all implementations).
* size(): Returns the number of elements in the set (might not be available in all implementations).
* clear(): Removes all elements from the set.
* iterator(): Returns an iterator object to iterate through the elements in the set (might not be available in all implementations).
* toArray(): Converts the set to an array (might not be available in all implementations).
* addAll(collection): Adds all elements from another collection to the set (avoiding duplicates based on hash code and equals method).
* removeAll(collection): Removes all elements from the set that are also present in the specified collection (based on hash code and equals method).
* retainAll(collection): Retains only the elements in the set that are also present in the specified collection (based on hash code and equals method).
* containsAll(collection): Checks if the set contains all elements from the specified collection (based on hash code and equals method).

Set Methods (TreeSet):

* All methods from HashSet, plus methods for set navigation (first, last, etc.).

Map Methods (HashMap):

* put(key, value): Adds a key-value pair to the map.
* get(key): Retrieves the value associated with a key from the map.
* containsKey(key): Checks if the map contains a specific key and returns a boolean value.
* remove(key): Removes the key-value pair associated with a specific key from the map.
* isEmpty(): Checks if the map is empty and returns a boolean value (might not be available in all implementations).
* size(): Returns the number of key-value pairs in the map (might not be available in all implementations).
* clear(): Removes all key-value pairs from the map.
* keySet(): Returns a set view of all the keys in the map (might not be available in all implementations).
* values(): Returns a collection view of all the values in the map (might not be available in all implementations).
* entrySet(): Returns a set view of all the key-value pairs as Map.Entry objects (might not be available in all implementations).
* putAll(map): Copies all key-value pairs from another map to this map (might overwrite existing keys).
* getOrDefault(key, defaultValue): Retrieves the value associated with a key, or returns a default value if the key is not found.
* replace(key, oldValue, newValue): Replaces the value associated with a key only if the current value is equal to the oldValue.
* computeIfAbsent(key, mappingFunction): If the key is not present, computes a value based on the provided mapping function and adds it to the map.

Map Methods (TreeMap):

* All methods from HashMap, plus methods for map navigation (first key, last key, etc.).

Tree Methods (Binary Search Tree, etc.):

* add(element): Inserts an element into the tree.
* contains(element): Checks if the tree contains a specific element and returns a boolean value.
* remove(element): Removes an element from the tree.
* inOrder(), preOrder(), postOrder(): Methods for tree traversal (iterative or recursive).
* preOrder(): Pre-order traversal (might be iterative or recursive).
* postOrder(): Post-order traversal (might be iterative or recursive).
* isEmpty(): Checks if the tree is empty and returns a boolean value (Optional).
* size(): Returns the number of elements in the tree (Optional).
* height(): Returns the height of the tree (Optional, might be specific to Binary Search Trees).
* getRoot(): Returns the root node of the tree (Optional).

Graph Methods:

* Adjacency List representation: HashMap<Node, List<Node>> to store neighbors for each node.
* addNode(Node node) (Optional): Adds a new node to the graph (might vary depending on the implementation).
* addEdge(Node source, Node destination): Creates a directed edge from the source node to the destination node (might need an additional method for undirected edges).
* getNeighbors(Node node): Retrieves the list of neighbor nodes for a specific node (utilizes the HashMap structure).
* BFS (Breadth-First Search): Explores the graph in a level-by-level manner.
* DFS (Depth-First Search): Explores the graph along a single path as far as possible before backtracking.
* isConnected(): Checks if the graph is connected (all nodes are reachable from each other) (Optional).
* shortestPath(Node source, Node destination): Finds the shortest path between two nodes in the graph (Optional).
* hasCycle(): Checks if the graph contains a cycle (a loop of connected edges) (Optional).

Math Library Methods:

* Math.abs(x): Returns the absolute value of a number.
* Math.max(x, y) / Math.min(x, y): Returns the maximum/minimum of two numbers.
* Math.ceil(x) / Math.floor(x): Rounds a number up/down to the nearest integer.
* Math.round(x): Rounds a number to the nearest integer.
* Math.sqrt(x): Returns the square root of a number.
* Math.pow(x, y): Raises a number to a power.
* Math.sin(x) / Math.cos(x) / Math.tan(x): Trigonometric functions (input in radians).
* Math.random(): Generates a pseudo-random number between 0 (inclusive) and 1 (exclusive).
* Math.PI: Mathematical constant pi (approximately 3.14159).
* Math.E: Mathematical constant e (approximately 2.71828).
* Math.exp(x): Returns the exponent of x (e raised to the power of x).
* Math.log(x): Returns the natural logarithm (base e) of a number.
* Math.log10(x): Returns the base-10 logarithm of a number.

Collections Class Methods:

* Collections.sort(List list): Sorts the elements in a list.
* Collections.binarySearch(List list, Object key): Performs binary search on a sorted list.
* Collections.reverse(List list): Reverses the order of elements in a list.
* Collections.shuffle(List list): Randomly shuffles the elements in a list.
* Collections.fill(List list, Object obj): Replaces all elements in a list with a specified object.
* Collections.copy(List destList, List srcList): Copies all elements from one list to another.
* Collections.min(Collection col) / Collections.max(Collection col): Returns the minimum or maximum element from a collection.
* Collections.frequency(Collection col, Object obj): Counts the number of occurrences of an element in a collection.
* Collections.emptySet() (Optional): Returns an empty, immutable Set.
* Collections.singleton(Object obj) (Optional): Returns a singleton List containing a single element.
* Collections.unmodifiableCollection(Collection col) (Optional): Returns an unmodifiable view of the given collection.
* Collections.addAll(Collection destCol, Collection srcCol) (Optional): Adds all elements from one collection to another collection.

Wrapper Class Methods:

**Number (Superclass for most numeric wrappers):**

* static valueOf(String s): Parses a string representation of a number and returns an object of the corresponding wrapper class. (e.g., Integer.valueOf("10") returns an Integer object with the value 10). There are variations of this method for different radix (base) conversions.
* toString(): Converts the object back to its string representation. (e.g., Integer.toString(10) returns the string "10").

Primitive Value Wrappers:

* intValue(): Returns the int value for Integer.
* doubleValue(): Returns the double value for Double.
* longValue(): Returns the long value for Long.
* floatValue(): Returns the float value for Float.
* equals(Object obj): Checks if this object is equal to another object. This compares the primitive values wrapped by the objects.
* compareTo(Number num): Compares this object with another Number object. The comparison happens based on the primitive values.

**Primitive Specific Wrapper Classes:**

**Integer:**

* static parseInt(String s, int radix): Parses a string representation of an integer in a specific radix (base). (e.g., Integer.parseInt("FF", 16) parses the hexadecimal string "FF" and returns 255).
* MAX\_VALUE: The highest possible value for an int (2147483647).
* MIN\_VALUE: The lowest possible value for an int (-2147483648).

**Long:**

* static parseLong(String s, int radix): Similar to Integer.parseInt but for long values.
* MAX\_VALUE: The highest possible value for a long (9223372036854775807).
* MIN\_VALUE: The lowest possible value for a long (-9223372036854775808).

**Double:**

* static isNaN(double d): Checks if a double value is Not-a-Number (NaN).
* static isInfinite(double d): Checks if a double value is positive or negative infinity.
* MAX\_VALUE: The highest possible positive value for a double (represented by Double.POSITIVE\_INFINITY).
* MIN\_VALUE: The lowest possible negative value for a double (represented by Double.NEGATIVE\_INFINITY).

**Float:**

* static parseFloat(String s): Parses a string representation of a float.
* MAX\_VALUE: The highest possible positive value for a float (represented by Float.POSITIVE\_INFINITY).
* MIN\_VALUE: The lowest possible negative value for a float (represented by Float.NEGATIVE\_INFINITY).

**Boolean:**

* static parseBoolean(String s): Parses a string ("true" or "false") and returns a Boolean object.
* booleanValue(): Returns the primitive boolean value wrapped by the object.
* TRUE: Represents the true value.
* FALSE: Represents the false value.

**Character:**

* charValue(): Returns the primitive character value wrapped by the object.
* static isDigit(char ch): Checks if a character is a digit (0-9).
* static isLetter(char ch): Checks if a character is a letter (a-z or A-Z).
* static isWhitespace(char ch): Checks if a character is whitespace (space, tab, newline, etc.).
* There are many more methods for character classification (uppercase, lowercase, etc.) available in the Character class.
* MIN\_VALUE: The lowest possible value for a char (represented by Character.MIN\_VALUE).
* MAX\_VALUE: The highest possible value for a char (represented by Character.MAX\_VALUE).

**Additional Notes:**

* All wrapper classes except Character inherit common methods from Number like toString(), equals(), and compareTo().
* Wrapper classes are immutable, meaning their values cannot be changed after creation.
* Autoboxing and unboxing are automatic conversions between primitive types and their corresponding wrapper objects in Java.