**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()).

**ArrayList Methods:**

* add(element): Adds the specified element to the end of the list.
* add(index, element): Inserts the specified element at the specified position in the list.
* remove(index): Removes the element at the specified position in the list.
* remove(element): Removes the first occurrence of the specified element from the list.
* set(index, element): Replaces the element at the specified position with the specified element.
* get(index): Returns the element at the specified position in the list.
* clear(): Removes all elements from the list.
* contains(element): Returns true if the list contains the specified element.
* indexOf(element): Returns the index of the first occurrence of the specified element.
* lastIndexOf(element): Returns the index of the last occurrence of the specified element.
* size(): Returns the number of elements in the list.
* isEmpty(): Returns true if the list contains no elements.
* subList(fromIndex, toIndex): Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.
* toArray(): Returns an array containing all of the elements in this list in proper sequence.
* iterator(): Returns an iterator over the elements in the list in proper sequence.

**Stack Methods:**

* push(element): Pushes an element onto the top of the stack.
* pop(): Removes and returns the top element of the stack.
* peek(): Returns the top element of the stack without removing it.
* isEmpty(): Returns true if the stack contains no elements.
* search(element): Returns the 1-based position from the top of the stack where the specified element is located.

**Queue Methods:**

* add(element): Inserts the specified element into the queue.
* offer(element): Inserts the specified element into the queue (similar to add()).
* remove(): Retrieves and removes the head of the queue.
* poll(): Retrieves and removes the head of the queue, or returns null if the queue is empty.
* element(): Retrieves, but does not remove, the head of the queue.
* peek(): Retrieves, but does not remove, the head of the queue, or returns null if the queue is empty.
* isEmpty(): Returns true if the queue contains no elements.

**Set Methods (HashSet, TreeSet):**

* add(element): Adds the specified element to the set if it is not already present.
* remove(element): Removes the specified element from the set if it is present.
* contains(element): Returns true if the set contains the specified element.
* clear(): Removes all elements from the set.
* size(): Returns the number of elements in the set.
* isEmpty(): Returns true if the set contains no elements.
* iterator(): Returns an iterator over the elements in the set.
* toArray(): Returns an array containing all of the elements in the set.

**Map Methods (HashMap, TreeMap):**

* put(key, value): Associates the specified value with the specified key in the map.
* get(key): Returns the value to which the specified key is mapped, or null if the map contains no mapping for the key.
* remove(key): Removes the mapping for the specified key from the map if present.
* containsKey(key): Returns true if the map contains a mapping for the specified key.
* containsValue(value): Returns true if the map maps one or more keys to the specified value.
* clear(): Removes all of the mappings from the map.
* size(): Returns the number of key-value mappings in the map.
* isEmpty(): Returns true if the map contains no key-value mappings.
* keySet(): Returns a set view of the keys contained in the map.
* values(): Returns a collection view of the values contained in the map.
* entrySet(): Returns a set view of the mappings contained in the map.
* putAll(map): Copies all of the mappings from the specified map to this map.

**Math Methods:**

* abs(value): Returns the absolute value of a value.
* max(a, b): Returns the greater of two values.
* min(a, b): Returns the smaller of two values.
* sqrt(value): Returns the square root of a value.
* pow(base, exponent): Returns the value of the first argument raised to the power of the second argument.
* round(value): Returns the closest long or int to the argument.
* floor(value): Returns the largest (closest to positive infinity) double value that is less than or equal to the argument and is equal to a mathematical integer.
* ceil(value): Returns the smallest (closest to negative infinity) double value that is greater than or equal to the argument and is equal to a mathematical integer.
* random(): Returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0.
* log(value): Returns the natural logarithm (base e) of a value.
* log10(value): Returns the base 10 logarithm of a value.
* exp(value): Returns Euler's number e raised to the power of a value.
* sin(angle): Returns the trigonometric sine of an angle.
* cos(angle): Returns the trigonometric cosine of an angle.
* tan(angle): Returns the trigonometric tangent of an angle.
* asin(value): Returns the arc sine of a value; the returned angle is in the range -pi/2 through pi/2.
* acos(value): Returns the arc cosine of a value; the returned angle is in the range 0.0 through pi.
* atan(value): Returns the arc tangent of a value; the returned angle is in the range -pi/2 through pi/2.

**Other Useful Methods:**

* **Collections Class**:
  + sort(list): Sorts the specified list into ascending order according to the natural ordering of its elements.
  + binarySearch(list, key): Searches the specified list for the specified key using the binary search algorithm.
  + reverse(list): Reverses the order of the elements in the specified list.
  + shuffle(list): Randomly permutes the specified list using a default source of randomness.
  + copy(dest, src): Copies all of the elements from one list into another.
  + fill(list, obj): Replaces all of the elements in the specified list with the specified element.
  + frequency(collection, obj): Returns the number of elements in the specified collection equal to the specified object.
  + disjoint(c1, c2): Returns true if the two specified collections have no elements in common.

**Graph Methods:**

* **Graph Representation**:
  + **Adjacency Matrix**: A 2D array used to represent a graph, where the element at row i and column j represents the edge between vertex i and vertex j.
  + **Adjacency List**: An array of lists, where the index represents the vertex and each element in the list represents the other vertices connected to that vertex.
* **Common Graph Algorithms**:
  + DFS (Depth-First Search): A traversal algorithm that starts at the root (or any arbitrary node) and explores as far as possible along each branch before backtracking.
  + BFS (Breadth-First Search): A traversal algorithm that starts at the root (or any arbitrary node) and explores all neighbor nodes at the present depth before moving on to nodes at the next depth level.
  + Dijkstra's Algorithm: An algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks.
  + Bellman-Ford Algorithm: An algorithm for finding the shortest paths from a single source vertex to all other vertices in a weighted graph.
  + Floyd-Warshall Algorithm: An algorithm for finding shortest paths in a weighted graph with positive or negative edge weights (but with no negative cycles).

**Graph Data Structure Operations**:

* addVertex(vertex): Adds a vertex to the graph.
* addEdge(vertex1, vertex2): Adds an edge between vertex1 and vertex2.
* removeVertex(vertex): Removes a vertex from the graph.
* removeEdge(vertex1, vertex2): Removes the edge between vertex1 and vertex2.
* getAdjacentVertices(vertex): Returns a list of all vertices adjacent to the given vertex.
* hasVertex(vertex): Checks if the graph contains the specified vertex.
* hasEdge(vertex1, vertex2): Checks if there is an edge between vertex1 and vertex2.

**Wrapper Class Methods**:

Wrapper classes provide a way to use primitive data types (int, boolean, etc.) as objects. The wrapper classes are part of the java.lang package.

* **Common Wrapper Classes**:
  + Boolean: Wraps a value of the primitive type boolean in an object.
  + Character: Wraps a value of the primitive type char in an object.
  + Byte: Wraps a value of the primitive type byte in an object.
  + Short: Wraps a value of the primitive type short in an object.
  + Integer: Wraps a value of the primitive type int in an object.
  + Long: Wraps a value of the primitive type long in an object.
  + Float: Wraps a value of the primitive type float in an object.
  + Double: Wraps a value of the primitive type double in an object.

**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.
  + Example: 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.
  + Example: 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).
  + Example: 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.).
* 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.