**JAVA NOTES - HG**

|  |  |
| --- | --- |
| Content | Link |
| Abstract class vs interface | [Abstract class vs interface](#AbstractClassVSInterface) |
| Abstraction VS Encapsulation | [Abstraction VS Encapsulation](#AbstractionvsEncapsulation) |
| Access Modifiers | [Access Modifiers](#AccessModifiers) |
| Array | [Array](#Array) |
| Array List | [Array List](#ArrayList) |
| Array List Duplicates | [Array List Duplicates](#DuplicatesInArrayList) |
| Array Methods | [Array Methods](#ArrayMethods) |
| Array vs ArrayList | [Array vs ArrayList](#ArrayVSArrayList) |
| ArrayList Constructors | [ArrayList Constructors](#ArrayListConstructors) |
| ArrayList Methods | [ArrayList Methods](#ArrayListMethods) |
| Class | [Class](#ObjectAndClass) |
| Collections.sort() Method | [Collections.sort() Method](#CollectionsSortMethod) |
| Abstract Concept of Abstraction | [Abstract Concept of Abstraction](#ConceptOfAbstraction) |
| Constructor vs Method | [Constructor vs Method](#ConstructorvsMethod) |
| Error vs Exception | [Error vs Exception](#ErrorVsException) |
| Exception Handling | [Exception Handling](#ExceptionHandling) |
| final finally finalize | [final finally finalize](#finalfinallyfinalize) |
| Hashtable vs HashMap | [Hashtable vs HashMap](#HashtableVsHashMap) |
| LIST - SET - MAP when to use? | [LIST - SET - MAP when to use?](#WhenUseListSetMap) |
| List Set Map Differences | [List Set Map Differences](#DiffListSetMap) |
| List vs Set | [List vs Set](#ListVsSet) |
| Object | [Object](#ObjectAndClass) |
| Object vs Class | [Object vs Class](#ObjectAndClass) |
| Overloading vs Overriding | [Overloading vs Overriding](#OverloadingVsOverriding) |
| Polymorphism vs Inheritance | [Polymorphism vs Inheritance](#PolymrphsmVSInhrtnce) |
| Propagate | [Propagate](#Propagate) |
| String Methods | [String Methods](#StringMethods) |
| String Reversing using for loop | [String Reversing using for loop](#StringReversing) |
| String vs StringBuilder | [String vs StringBuilder](#StringVsStringBuilder) |
| StringBuffer and StringBuilder | [StringBuffer and StringBuilder](#StringBufferStringBuilder) |
| StringBuilder | [StringBuilder](#StringBuilder) |
| StringBuilder Constructors | [StringBuilder Constructors](#StringBuilderConstructors) |
| StringBuilder Methods | [StringBuilder Methods](#StringBuilderMethods) |
| throw vs throws | [throw vs throws](#ThrowVsthrows) |

**JAVA**

JVM is an abstract machine, JVM runs the java byte code.

JRE is what you need to run a java program. JRE contains set of libraries and other files that JVM uses at runtime.

JRE = JVM + Library classes

JDK is what you need to compile Java source code and contains JRE development tools.

JDK = JRE + Development tools

**Object Oriented Programming (OOP)**

OOP is a programming language model organized around an object rather than actions.

* It makes the development and maintenance easier
* It provides the data hiding
* It provides ability to simulate real world

***OOP language must follow 4 principles***

1. ***Encapsulation***

We can hide direct access to the data by using private keyword and we can access private that that by using get there and set of methods.

1. ***Abstraction***

Abstraction is a process of hiding implementation details and showing only the functionality to the user. It lets you focus on what the object does instead of how.

1. ***Inheritance***

Inheritance is used to define the relationship between two classes. When a child class acquires all the properties and behaviors of parent class known as inheritance. Child class can reuse all the codes written in parent class.

Inheritance provides code reusability.

1. ***Polymorphism***

Polymorphism is an ability of an object to behave in multiple forms. The most common use of polymorphism in Java; when a parent class reference type of variable is used to refer to a child class object.

*for example:*

WebDriver driver = new ChromeDriver();

We use method overloading and overriding to achieve polymorphism.

What is the concept of Abstraction?

In OOP Abstraction is a process of hiding implementation details from the user, only the functionality will be provided to the user. In other words, the user will have the information on what the object does instead of how it does.

\*\*\* In Java abstraction is achieved by using Abstract classes and Interfaces \*\*\*

*for example:*

When login to your bank account online, you enter your user ID and password and hit login button. What happens when you press login, how the input that sent to the server, how it gets verified is all abstracted away a from you.

**Abstract Class:** A class that is declared with abstract keyboard is known as abstract class.

It can have abstract and non-abstract class. When we want to use abstract class, we use *extend* keyword.

Cannot be instantiated and cannot create an instance of an Abstract Class

**Interface:** Interface is a blueprint of a class. It is a template and it's declared with *interface* keyword.

Methods: Interface can have abstract, default, and static methods

Variables: Interface can have public final static variables.

When we want to use interface, we use *implement* keyword.

Interface cannot be instantiated and cannot create an instance of Interface.

Abstract Class vs Interface

|  |  |
| --- | --- |
| **Abstract Class** | **Interface** |
| A class can extend only one abstract class. Multiple inheritance is not possible using abstract class | Any class can implement multiple interfaces. Multiple inheritance is possible using interfaces in Java |
| Can have constructor | Cannot have constructor |
| Abstract class can also have private, protected fields. | Fields in Interface are public static final |
| Abstract class may contain both abstract and concrete methods. | Up to java 7, interface can only contain public abstract methods. We cannot declare any concrete methods inside interface. |

Abstraction vs Encapsulation?

Abstraction lets you focus on what the object does instead of how it does.

Encapsulation means; hiding the internal details of how the object does something.

Abstraction is used for hiding the unwanted data and giving relevant data.

Encapsulation means hiding the code and data as well as protect the data from outside.

Abstraction can be achieved by using abstract class and interfaces.

And capsulation can be achieved by using private keyword.

Polymorphism vs Inheritance

Like in real world inheritance is used to define the relationship between two classes. It is similar to father-son relationship. In Java we have parent class (a.k.a. super class) and child class (a.k.a. subclass) similar to the real world, child inherits parents qualities (methods, codes).

A child class can reuse all the codes written in parent class and only write code for behavior which is different than the parent class.

Inheritance is meant for code-reuse.

On the other hand, polymorphism is an ability of an object to behave in multiple forms.

Polymorphism is classified as overloading and overriding. Polymorphism and Inheritance are related to each other because Inheritance what makes polymorphism possible. Without any relationship between two classes it is not possible to write polymorphic code.

**Access Modifiers: private, default, protected, public**

Access modifiers used for determining the accessibility of class, method and variable.

Total **4** access modifiers:

* **private** only accessible within the same class *(class private)*
* **default** is accessible only within the same package *(package private – no default keyword used!)*
* **protected** accessible only within the same package and, also child class outside the package. *(package private + child class)*
* **public** accessible anywhere.

**Note: 1.** Top *level classes* can't be **private** and **protected**

2. *Local variables* can't be **public, private** and **protected**

|  |  |
| --- | --- |
| **Constructor (); VS Method () {}** | |
| **Constructor ();** | **Method () {}** |
| **Java provides a default constructor (if user did not create one)** | **Java doesn't provide a method** |
| **Constructor name must be same as class name** | **Name can be same as class name or different** |
| **Has No return type** | **Has return** |
| **Constructors Invoked implicitly** | **Methods invoked explicitly** |
| **Can't be inherited by a child/sub class** | **Can be Inherited by child/sub class** |
| **Called automatically when a new object is created.** | --- |
| **Can't be private** | **Can be private** |

---

|  |  |
| --- | --- |
| **Overloading VS Overriding** | |
| **Overloading** | **Overriding** |
| **Same Method Name Different Parameters** | **Same Name Same Parameters** |
| **Occurs in the same class** | **Occurs in different related classes** |
| **CAN Overload *Static, final* and *private* methods** | **CAN NOT Override *Static, final* and *private* methods** |
| **Return type CAN be same or different** | **Return type MUST be SSAME or COVARIANT** |

*\*\*\** ***Three important Interface of Java Collection framework: List, Set and Map*** *\*\*\**

|  |  |
| --- | --- |
| **Difference between LIST - SET - MAP** | |
| **LIST** | **Provides Ordered and Indexed Collection which CAN contain Duplicates.**  **Extends Collection Interface** |
| **SET** | **Provides Un-Ordered Collection of Unique Objects. NO Duplicates Allowed!**  **Extends Collection Interface** |
| **MAP** | **Provides a Data Structure based on Key and Value.**  **Key must be unique (like user name) Value can be duplicate or anything** |

\*\*\* ***List and Set are both Interface and are extend Collection Interface*** \*\*\*

***But what is the difference between List and Set?***

|  |  |  |  |
| --- | --- | --- | --- |
| **LIST** | | **SET** | |
| **√** | Allows Duplicates | **X** | No Duplicates Allowed |
| **√** | Maintains Order | **X** | Doesn't maintain any order |

**Finding Duplicates in an Array List:**

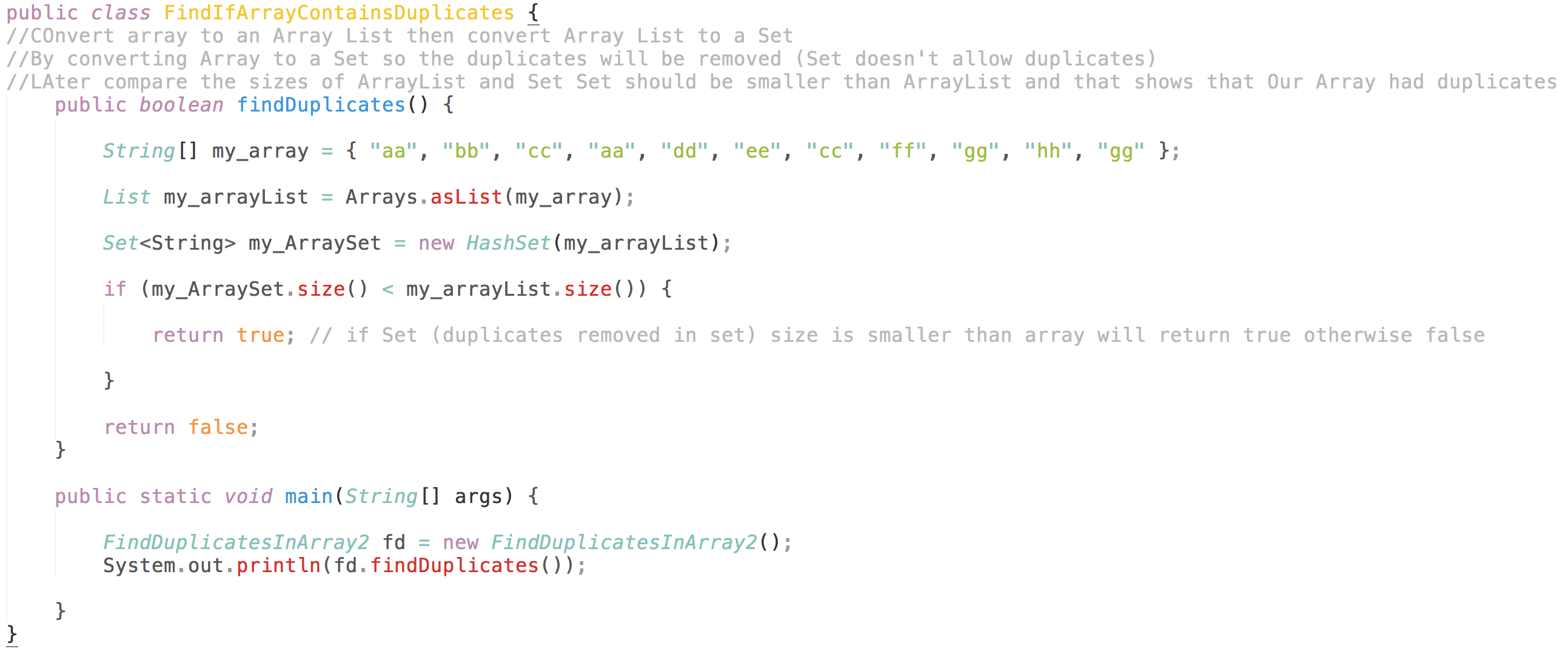
**Quick way to find if an Array List contains duplicates or not is to convert that array into a SET!**

**Since Set doesn't allow duplicates Set will be smaller than the original Array/ArrayList (if contains duplicates)**

**Use .length() method for Array when determining the size of an Array.**

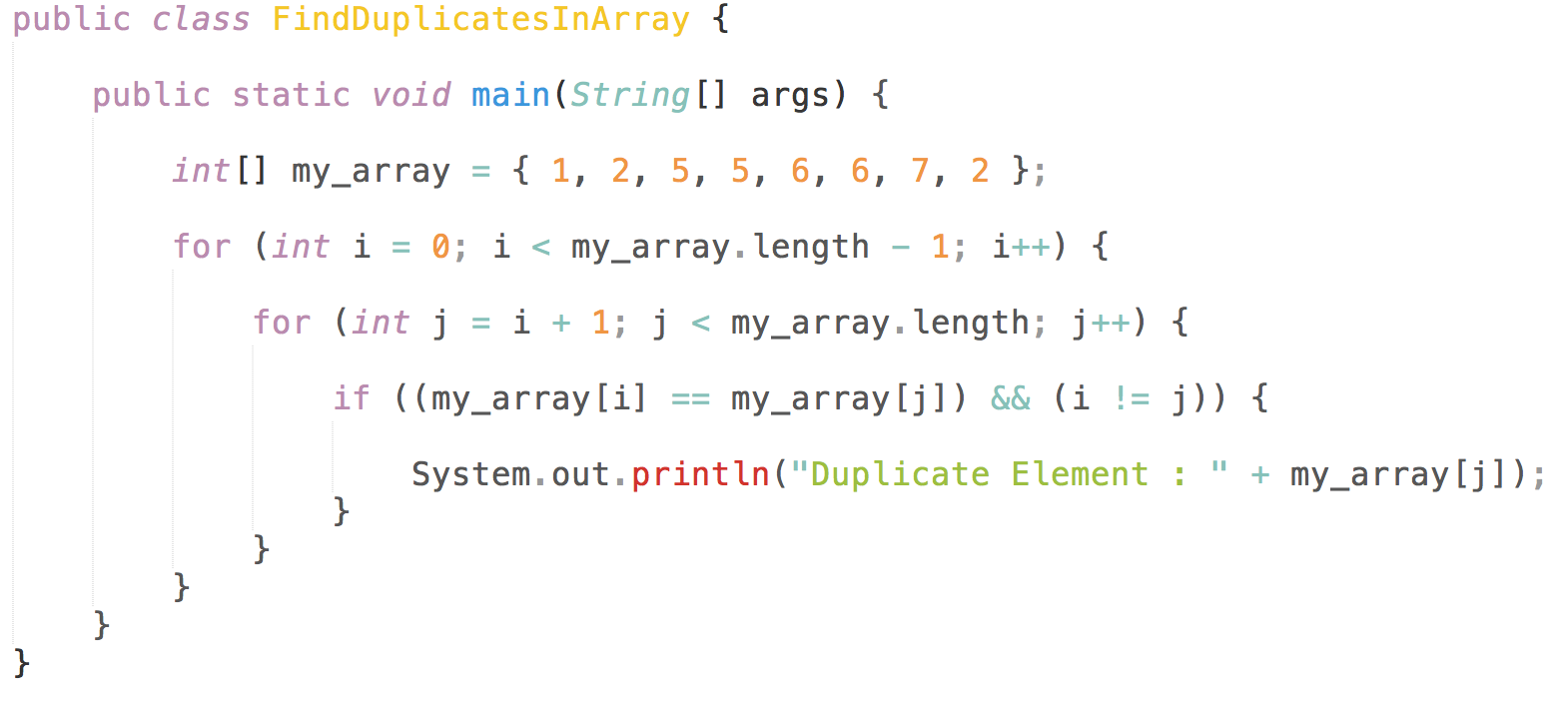
**Use .size() method for an ArrayList when determining the size of an ArrayList.**

***Example:***

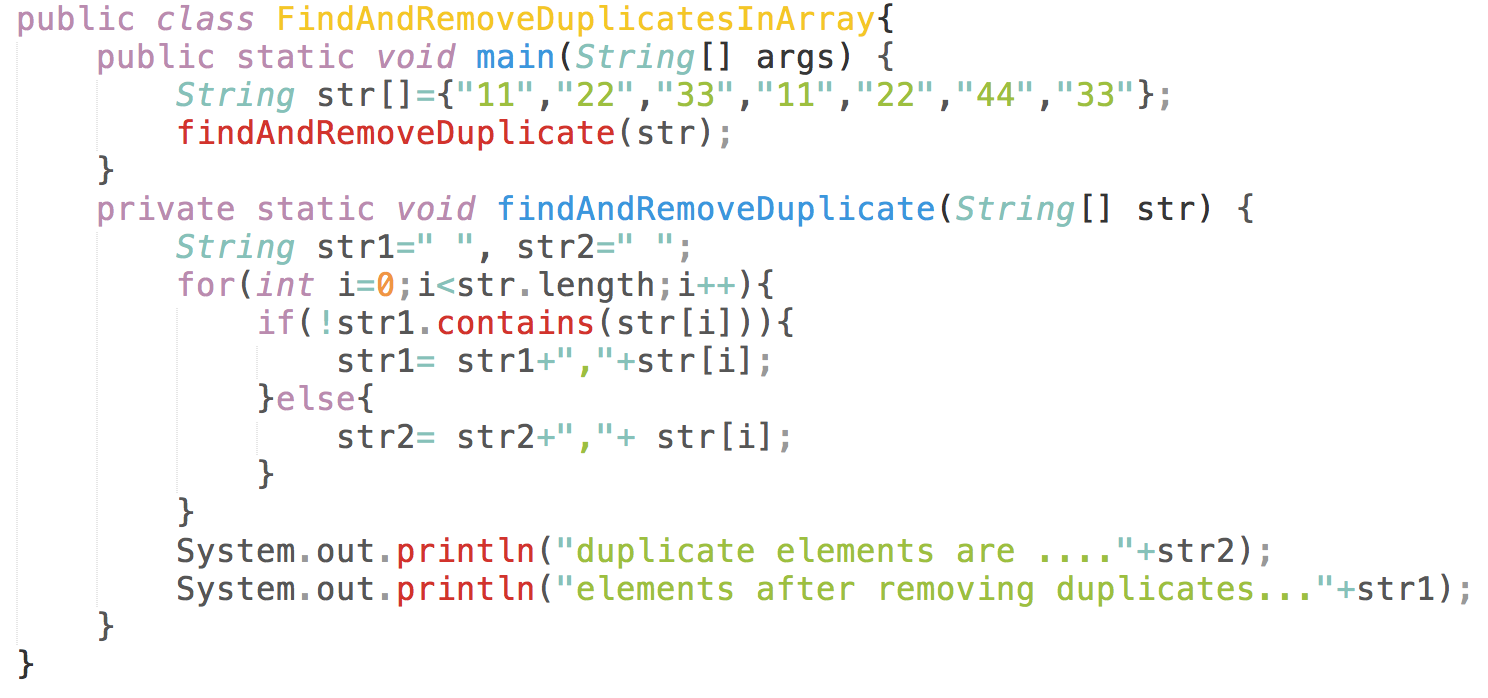
****

**Do the same by using for loop:**

Compare each element of Array to each other and return true if any duplicate found.

****

***Find Duplicates in Array and remove the Duplicates by using for loop:***

****

|  |  |
| --- | --- |
| **Array VS ArrayList** | |
| **Array** | **ArrayList** |
| **Array is part of core Java programming and has special syntax** | **ArrayList is a part of Collection Framework and Implements List Interface** |
| **Array is fixed length data structure. You CAN'T change the length of an Array once created. NOT Resizable.** | **ArrayList is resizable.** |
| **Array CAN contain both primitives and objects.** | **ArrayList CAN contain only Objects.** |
| **Array can be multi dimensional** | **ArrayList is single dimensional only** |
| **Length** | **Size()** |

*Since Java 5, primitives are automatically converted in objects which is known as auto-boxing.*

|  |  |
| --- | --- |
| **Hashtable VS HashMap** | |
| **Hashtable** | **HashMap** |
| **Hashtable is tread-safe** | **HashMap is NOT tread-safe.** |
| **Slow (since it is tread-safe)** | **Fast (since it is not tread-safe)** |
| **Both Hashtable and HashMap are implement Map Interface and both are stored "Key" and "Value"** | |

|  |  |
| --- | --- |
| **When to USE LIST - SET - MAP** | |
| **LIST** | **When you need to access elements frequently by using index use LIST.**  **ArrayList provides faster access if you know the index.** |
| **SET** | **When you want to store elements and want to maintain an order use SET.**  **List is an ordered collection and maintain order.** |
| **When you want to create a collection of unique elements and don't want any duplicates then choose any Set implementation (HashSet…….)** | |
| **When you want to store data in form "Key" and "Value" than Map is the way to go.**  **You can choose HashMap, HashTable..** | |

|  |  |
| --- | --- |
| **Error VS Exception** | |
| **Error** | **Exception** |
| **Error represents error which are generally CAN'T be handled. (OutOfMemory Error, No ClassFound Error)** | **Exception represent errors which can be catched and handled.**  **(IOException, NullPointerException)** |
| **Recovering from Error is NOT possible. The only solution is to terminate the execution**  ***Examples:***  java.lang.StackOverflowError  java.lang.OutOfMemoryError |  |

|  |  |
| --- | --- |
| **Throw VS throws** | |
| **throw** | **throws** |
| **"throw" keyword is used to throw an exception explicitly** | **"*throws"* used for declare an exception:**  **"*throws"* works similar to try-catch block** |
| **"throw" is followed by an instance of Exception Class**  exm: **throw new AritmeticException("Aritmetik Exception");** | **"throws" is followed by Exception Class Name**  **exm: throws AritmeticException;** |
| **"throw" keyword is used inside the method body to invoke an exception**  throw new Exception("You have some exception")  throw new IOException("Connection failed!!") | **"throws" clause is used in method declaration (signature)**  **to declare an exception**  public int myMethod() throws IOException, ArithmeticException, NullPointerException {} |
| **You cannot throw multiple exceptions with "throw"** | **You can declare multiple exceptions e.g. public void method()throws IOException, SQLException** |
| **checked exceptions CAN'T be propagated with "throw" only because it is explicitly used to throw an particular exception** | **Checked Exception can be propagated with "throws"** |
| **Propagate (extend/spread/yayilma):**  Uncaught exceptions are propagated in the call stack until stack becomes empty, this propagation is called Exception Propagation.  An exception propagates from method to method, up the call stack, until it's caught. So if a() calls b(), which calls c(), which calls d(), and if d() throws an exception, the exception will propagate from d to c to b to a, unless one of these methods catches the exception. | |

|  |
| --- |
| **How to handle Exceptions** |
| try-catch block: is placed around the code that might generate an exception. Every *try* block should be followed either by a  *catch* or *finally* block |
| finally block:follows a *try* or *catch* block. *finally* block always executed, no matter what. |
| throws keyword: if a method does not handle a *checked exception*, the method must declare it using the *throws* keyword.  The *throws* keyword appears at the end of Method's signature. |
| throw keyword: you can throw an exception by using the *throw* keyword inside the method body. |

|  |
| --- |
| **final vs finalize vs finally** |
| final is a keyword and used to apply restrictions on class, method and variable.   * **final** *Class* CAN'T be Inherited * **final** *Method* CAN'T be Overridden * **final** *Variable* value CAN'T be changed. |
| finallyis a block and used to place important code, it will be executed whether exception handled or not |
| finalize is a method and used to perform clean-up processing before Object is Garbage collected. It might be executed or not  depends on the compiler. |

|  |  |
| --- | --- |
| **Object VS Class** | |
| **Object** | **Class** |
| **Object is a *member* or *instance* of a Class** | **Class is a *blueprint* or *template* which you can create as many objects as you like.** |
| **Object is created by new keyword** | **Class is declared using Class keyword** |

[**String**](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html) **-** [**StringBuilder**](https://docs.oracle.com/javase/tutorial/java/data/buffers.html) **- StringBuffer**

|  |  |
| --- | --- |
| **StringBuffer VS StringBuilder** [link](http://net-informations.com/java/cjava/builder.htm) | |
| **StringBuffer** | **StringBuilder** |
| **StringBuffer is *Synchronized* (thread safe)** | **StringBuilder NOT synchronized (not thread safe)** |
| **Less efficient (slower)** | **More efficient (faster)** |

|  |  |
| --- | --- |
| **String VS StringBuilder** | |
| **String** | **StringBuilder** |
| **String is immutable**  *(means you CAN'T change the object itself, but can change the reference of the object)* | **StringBuilder (and StringBuffer) Mutable**  *(means* we can make changes to the value stored in the object*)* |

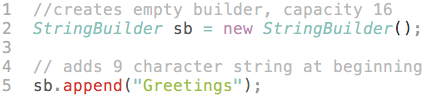
**StringBuilder Class:**

[StringBuilder](https://docs.oracle.com/javase/8/docs/api/java/lang/StringBuilder.html)objects are like [String](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html)objects, except that they can be modified. Internally, these objects are treated like variable-length arrays that contain a sequence of characters. At any point, the length and content of the sequence can be changed through method invocations.

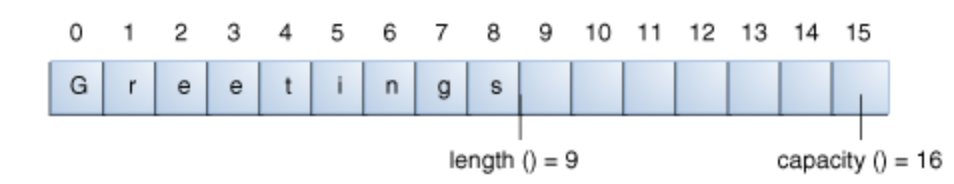
**NOTE:** *Strings should always be used unless string builders offer an advantage in terms of simpler code (*[*example link*](https://docs.oracle.com/javase/tutorial/java/data/buffers.html)*) or better performance. For example, if you need to concatenate a large number of strings, appending to a StringBuilder object is more efficient.*

|  |  |
| --- | --- |
| **StringBuilder Constructors** | |
| **Constructor** | **Description** |
| StringBuilder () | Creates an empty string builder with a capacity of 16 (16 empty elements). |
| StringBuilder (CharSequence cs) | Constructs a string builder containing the same characters as the specified CharSequence, plus an extra 16 empty elements trailing the CharSequence. |
| StringBuilder (int initCapacity) | Creates an empty string builder with the specified initial capacity. |
| StringBuilder (String s) | Creates a string builder whose value is initialized by the specified string, plus an extra 16 empty elements trailing the string. |

**Example: *Following code creates empty builder, capacity 16 and adds 9 character string at beginning and***

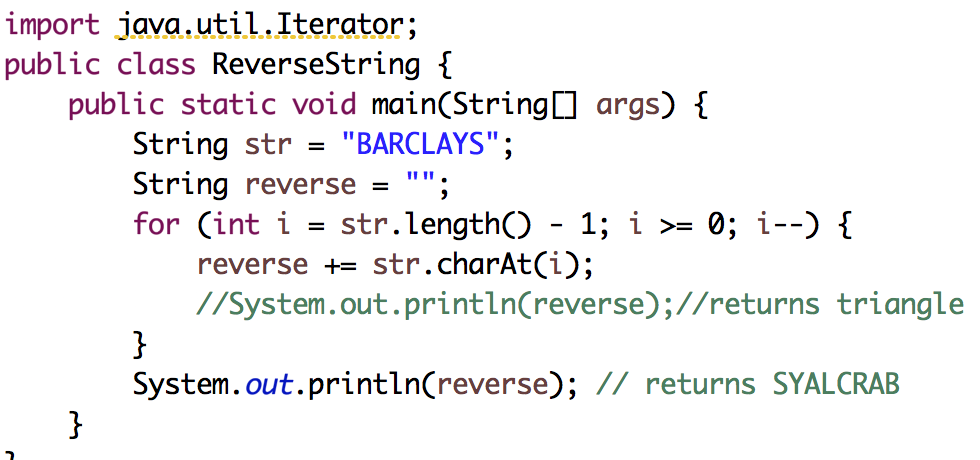
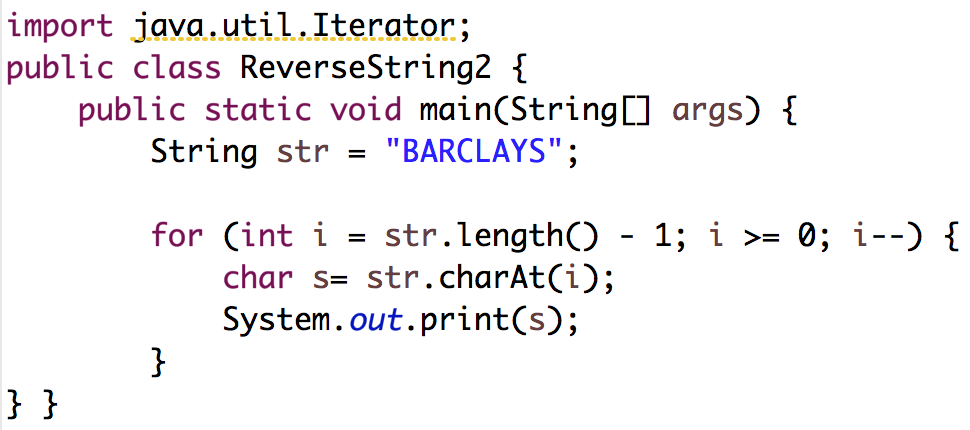
****

***will produce a string builder with a length of 9 and a capacity of 16:***

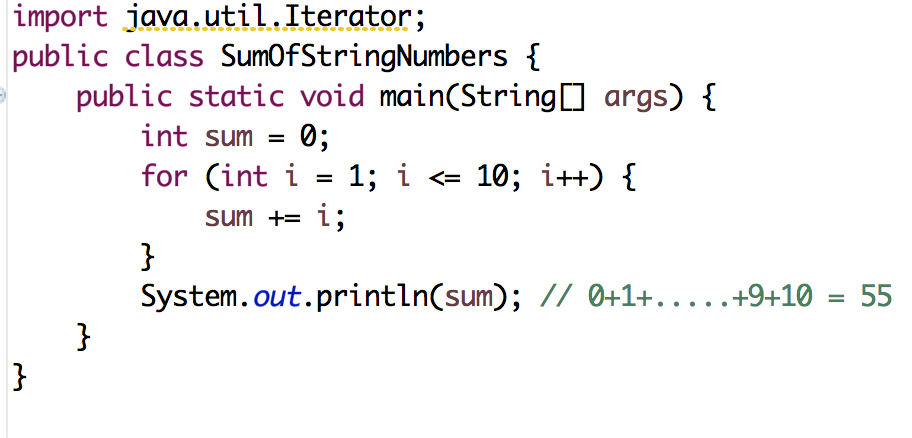
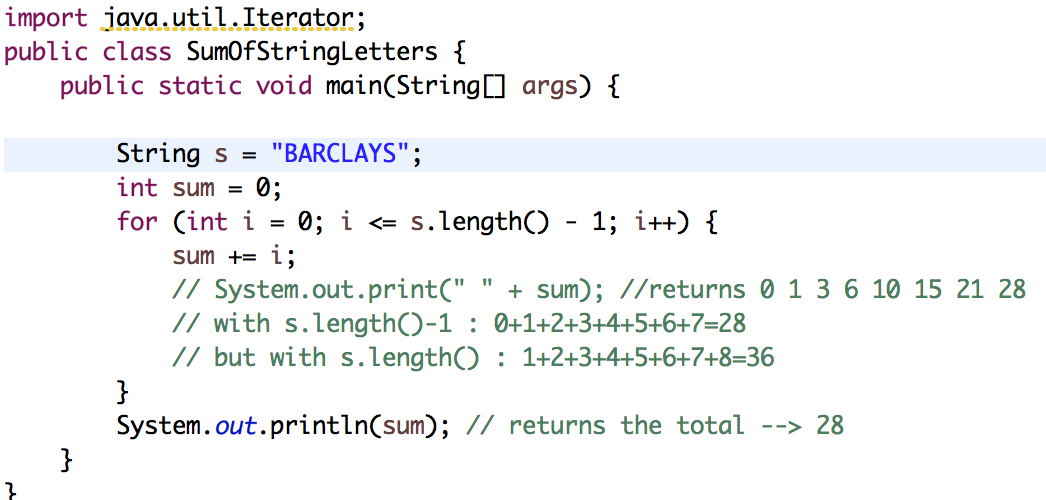


|  |
| --- |
| **StringBuilder Methods** |
| StringBuilder str = new StringBuilder("Hello"); Creates an Object in Heap Memory and contains the value of Hello |
| str.append("World"); 🡪 // returns Hello World |
| str.insert(1, ("World"); 🡪 // returns H**World**ello (World added to the specific field starting index no 1) |
| srt.replace(1,3, "World"); 🡪 // returns H**World**lo  (1 and 3 are not included) |
| str.delete(1,3); 🡪 // retuns Hlo  (1 and 3 are not included) |
| str.reverse(); 🡪 // retuns olleH |

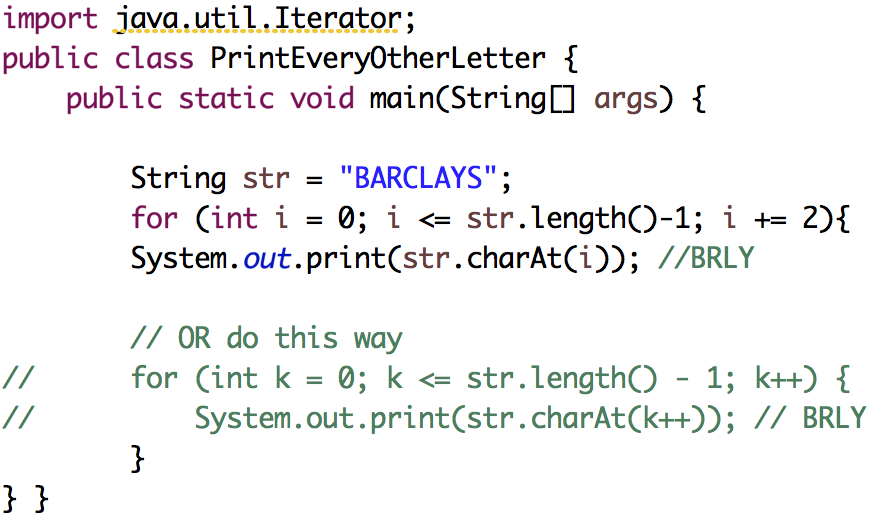
***String Reversing using for loop******:***

****

**Sum of String Letters Sum of String Numbers**

****

**Print every other String Letters**

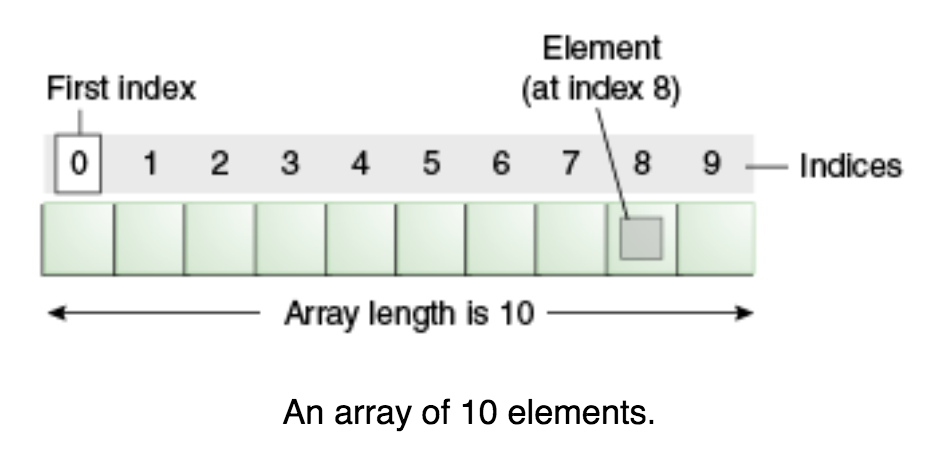
****

String Methods:

|  |  |
| --- | --- |
| **Method** | **Description** |
| [char charAt(int index)](https://www.javatpoint.com/java-string-charat) | returns char value for the particular index |
| [int length()](https://www.javatpoint.com/java-string-length) | returns string length |
| [String substring(int beginIndex)](https://www.javatpoint.com/java-string-substring) | returns substring for given begin index |
| [String substring(int beginIndex, int endIndex)](https://www.javatpoint.com/java-string-substring) | returns substring for given begin index and end index |
| [boolean contains(CharSequence s)](https://www.javatpoint.com/java-string-contains) | returns true or false after matching the sequence of char value |
| [boolean equals(Object another)](https://www.javatpoint.com/java-string-equals) | checks the equality of string with object |
| [boolean isEmpty()](https://www.javatpoint.com/java-string-isempty) | checks if string is empty |
| [String concat(String str)](https://www.javatpoint.com/java-string-concat) | concatinates specified string |
| [String replace(char old, char new)](https://www.javatpoint.com/java-string-replace) | replaces all occurrences of specified char value |
| [String replace(CharSequence old, CharSequence new)](https://www.javatpoint.com/java-string-replace) | replaces all occurrences of specified CharSequence |
| [static String equalsIgnoreCase(String another)](https://www.javatpoint.com/java-string-equalsignorecase) | compares another string. It doesn't check case. |
| [String[] split(String regex)](https://www.javatpoint.com/java-string-split) | returns splitted string matching regex |
| [String[] split(String regex, int limit)](https://www.javatpoint.com/java-string-split) | returns splitted string matching regex and limit |
| [String intern()](https://www.javatpoint.com/java-string-intern) | returns interned string |
| [int indexOf(int ch)](https://www.javatpoint.com/java-string-indexof) | returns specified char value index |
| [int indexOf(int ch, int fromIndex)](https://www.javatpoint.com/java-string-indexof) | returns specified char value index starting with given index |
| [int indexOf(String substring)](https://www.javatpoint.com/java-string-indexof) | returns specified substring index |
| [int indexOf(String substring, int fromIndex)](https://www.javatpoint.com/java-string-indexof) | returns specified substring index starting with given index |
| [String toLowerCase()](https://www.javatpoint.com/java-string-tolowercase) | returns string in lowercase. |
| [String toLowerCase(Locale l)](https://www.javatpoint.com/java-string-tolowercase) | returns string in lowercase using specified locale. |
| [String toUpperCase()](https://www.javatpoint.com/java-string-touppercase) | returns string in uppercase. |
| [String toUpperCase(Locale l)](https://www.javatpoint.com/java-string-touppercase) | returns string in uppercase using specified locale. |
| [String trim()](https://www.javatpoint.com/java-string-trim) | removes beginning and ending spaces of this string. |
| [static String valueOf(int value)](https://www.javatpoint.com/java-string-valueof) | converts given type into string. It is overloaded. |

Array

Comes from java.util class. An *array* is a container object that holds a fixed number of values of a single type. The length of an array is established when the array is created. After creation, its length is fixed. You have seen an example of arrays already, in the main method of the "Hello World!" application. This section discusses arrays in greater detail.



.size() method gives you the length of an Array.

Each item in an array is called an *element*, and each element is accessed by its numerical *index*. As shown in the preceding illustration, numbering begins with 0. The 9th element, for example, would therefore be accessed at index 8.

***Advantage of Java Array***

* **Code Optimization:** It makes the code optimized, we can retrieve or sort the data easily.
* **Random access:** We can get any data located at any index position.

***Disadvantage of Java Array***

* **Size Limit:** We can store only fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in java.
* [java.lang.Object](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html)
  + java.util.Arrays

You don’t need to import explicitly java.util.Arrays because it is already imported by default

Array Methods:

**Int [] myArray = {5,8,3}**

|  |  |
| --- | --- |
| **array.length**  **int length = myArray.length;** | if we wish to iterate to all the values in the array, the length of the Java array will determine how many times we need to loop |
| **Arrays.sort(myArray)** | Sortst "myArray" //import java.util.Arrays |
| **Arrays.binarySearch(myArray, valie);** | Will give the index of the "Value" Binary search works only if the Array sorted already. Doesn't work on unsorted array! |
| **String[][] myArray=new String [3][2]** | Creates two-dimensional array |
| **Arrays.asList(myArray);**  **List list1 = Arrays.asList(myArray); System.out.println(list1);** | Returns the list view of an Array. his method acts as bridge between array-based and collection-based APIs. |
| **Arrays.toString(myArray);**  for (int number : myArray) { System.out.println("Number = " + number);  }  System.out.println("The string representation of array is:"); System.out.println(Arrays.toString(myArray)); } } | returns a string representation of myArray |
| **Arrays.deepToString(myArray);** | returns a string representation of multi-dimensional array |

ArrayList:

The **java.util.ArrayList** class provides resizable-array and implements the **List** interface.

You must import **java.collection.ArrayList**  package in order to use ArrayList

Following are the important points about ArrayList:

* It implements all optional list operations and it also permits all elements, includes null.
* It provides methods to manipulate the size of the array that is used internally to store the list.
* The constant factor is low compared to that for the LinkedList implementation.

**.size() method gives the length of the Array List**

**ArrayList** **Class Constructors:**

|  |  |
| --- | --- |
| **ArrayList()** | This constructor is used to;  ***create an empty list with an initial capacity sufficient to hold 10 elements***. |
| **ArrayList(Collection<? extends E> c)** | This constructor is used to  ***create a list containing the elements of the specified collection***. |
| **ArrayList(int initialCapacity)** | This constructor is used to ***create an empty list with an initial capacity***. |

ArrayList Methods

|  |  |
| --- | --- |
| void | **add**(int index, Object element)  Inserts the specified element at the specified position in this ArrayList. |
| boolean | **add**(Object o)  Appends the specified element to the end of this ArrayList. |
| boolean | **addAll**(Collection c)  Appends all of the elements in the specified Collection to the end of this this ArrayList, in the  order that they are returned by the specified Collection's Iterator. |
| boolean | **addAll**(int index, Collection c)  Inserts all of the elements in the specified Collection into this ArrayList, starting at the specified position. |
| void | [clear()](https://courses.cs.washington.edu/courses/cse341/98au/java/jdk1.2beta4/docs/api/java/util/ArrayList.html#clear()) Removes all of the elements from this ArrayList. |
| [Object](https://courses.cs.washington.edu/courses/cse341/98au/java/jdk1.2beta4/docs/api/java/lang/Object.html) | [clone()](https://courses.cs.washington.edu/courses/cse341/98au/java/jdk1.2beta4/docs/api/java/util/ArrayList.html#clone()) Returns a shallow copy of this ArrayList. |
| boolean | **contains**(Object elem)  Returns true if this ArrayList contains the specified element. |
| void | [ensureCapacity(int minCapacity)](https://courses.cs.washington.edu/courses/cse341/98au/java/jdk1.2beta4/docs/api/java/util/ArrayList.html#ensureCapacity(int)) Increases the capacity of this ArrayList, if necessary, to ensure that it can hold at  least the number of components specified by the minimum capacity argument. |
| [Object](https://courses.cs.washington.edu/courses/cse341/98au/java/jdk1.2beta4/docs/api/java/lang/Object.html) | [get(int index)](https://courses.cs.washington.edu/courses/cse341/98au/java/jdk1.2beta4/docs/api/java/util/ArrayList.html#get(int)) Returns the element at the specified position in this ArrayList. |
| int | **indexOf**(Object elem)  Searches for the first occurence of the given argument, testing for equality using  the equals method |
| boolean | [isEmpty()](https://courses.cs.washington.edu/courses/cse341/98au/java/jdk1.2beta4/docs/api/java/util/ArrayList.html#isEmpty()) Tests if this ArrayList has no components. |
| int | lastIndexOf(Object elem)           Returns the index of the last occurrence of the specified object in this ArrayList. |
| [Object](https://courses.cs.washington.edu/courses/cse341/98au/java/jdk1.2beta4/docs/api/java/lang/Object.html) | [remove(int index)](https://courses.cs.washington.edu/courses/cse341/98au/java/jdk1.2beta4/docs/api/java/util/ArrayList.html#remove(int)) Removes the element at the specified position in this ArrayList. |
| [Object](https://courses.cs.washington.edu/courses/cse341/98au/java/jdk1.2beta4/docs/api/java/lang/Object.html) | **set**(int index, Object element)  Replaces the element at the specified position in this ArrayList with the specified  element |
| int | [size()](https://courses.cs.washington.edu/courses/cse341/98au/java/jdk1.2beta4/docs/api/java/util/ArrayList.html#size()) Returns the number of components in this ArrayList. |
| [Object[]](https://courses.cs.washington.edu/courses/cse341/98au/java/jdk1.2beta4/docs/api/java/lang/Object.html) | [toArray()](https://courses.cs.washington.edu/courses/cse341/98au/java/jdk1.2beta4/docs/api/java/util/ArrayList.html#toArray()) Returns an array containing all of the elements in this ArrayList in the correct order. |
| [Object[]](https://courses.cs.washington.edu/courses/cse341/98au/java/jdk1.2beta4/docs/api/java/lang/Object.html) | **toArray**(Object[] a)  Returns an array containing all of the elements in this ArrayList in the correct order. |
| void | [trimToSize()](https://courses.cs.washington.edu/courses/cse341/98au/java/jdk1.2beta4/docs/api/java/util/ArrayList.html#trimToSize()) Trims the capacity of this ArrayList to be the ArrayList's current size. |

# Java.util.Collections.sort() Method

**sort(List<T>)** method is used to sort the specified list into ascending order, according to the natural ordering of its element.

Declaration **of** java.util.Collections.sort() method: **public static <T extends Comparable<? super T>> void sort(List<T> list)**

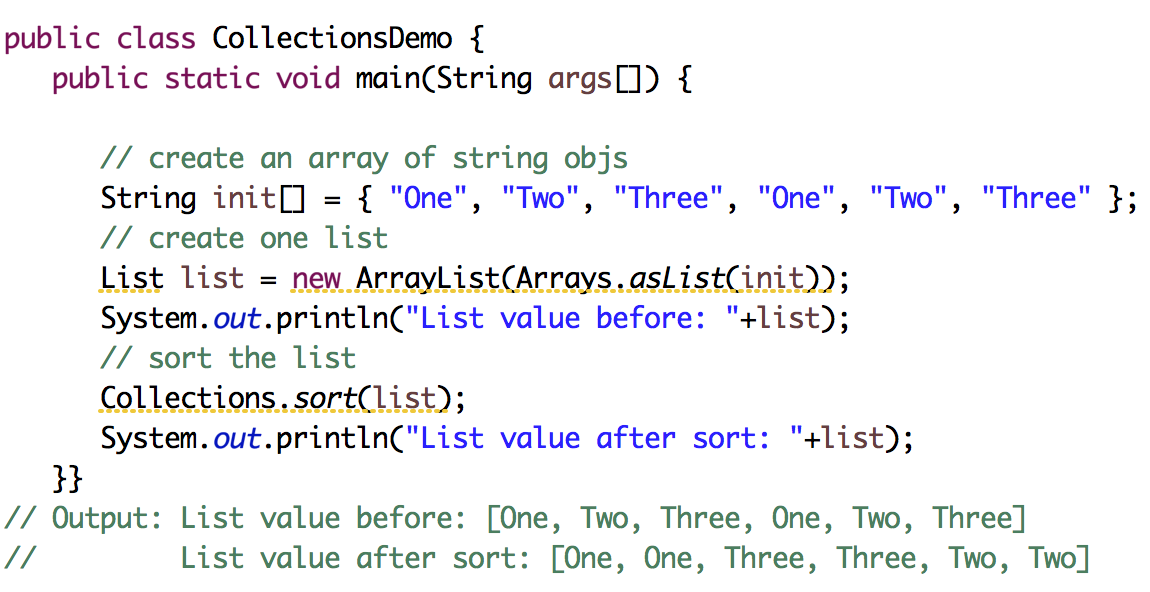
**Parameters:** ***list*** (This is the list to be sorted)

**Return Value:** NA

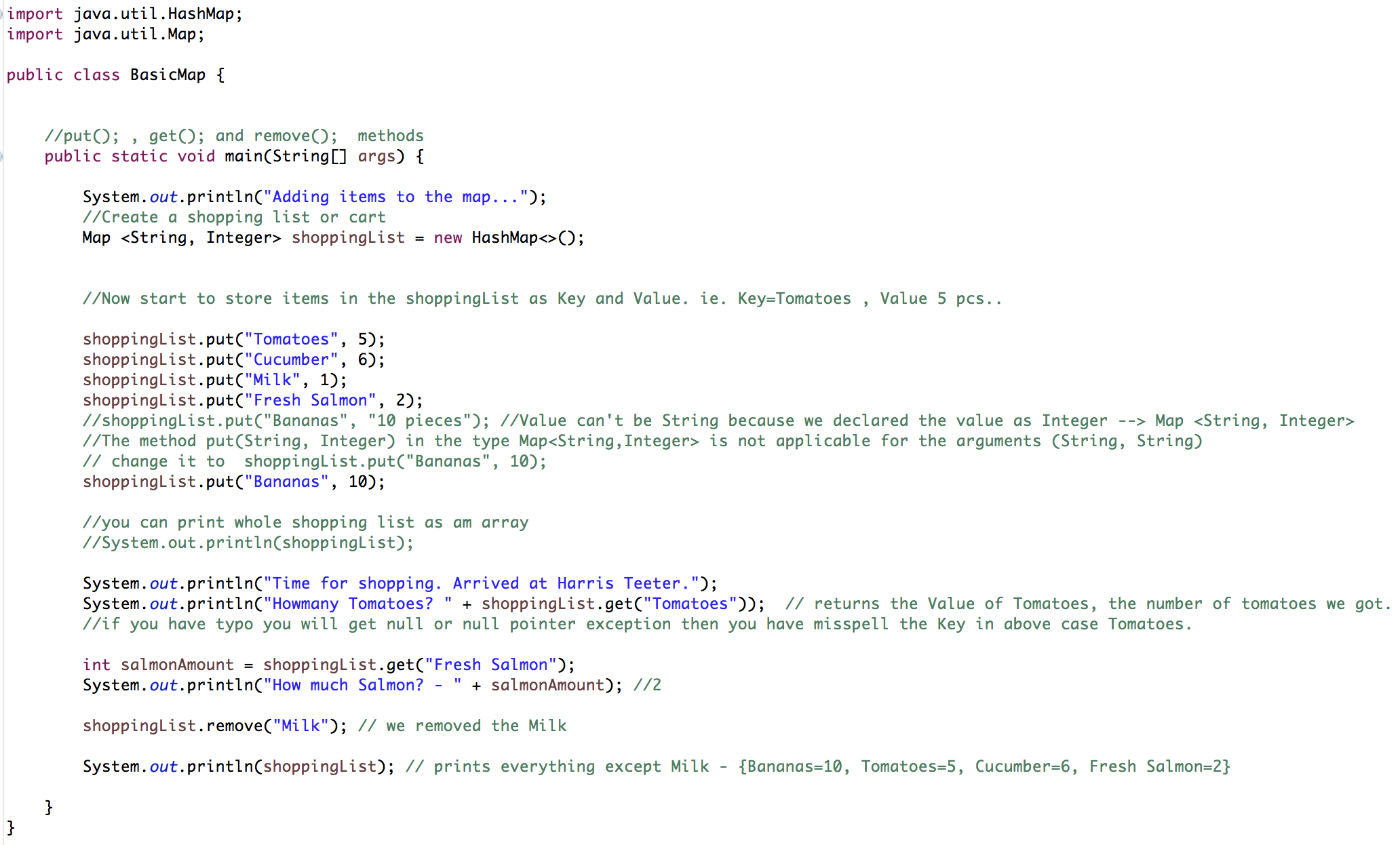
**Exceptions**

* **ClassCastException**: Throws if the list contains elements that are not mutually comparable (for example, strings and integers).
* **UnsupportedOperationException**: Throws if the specified list's list-iterator does not support the set operation.

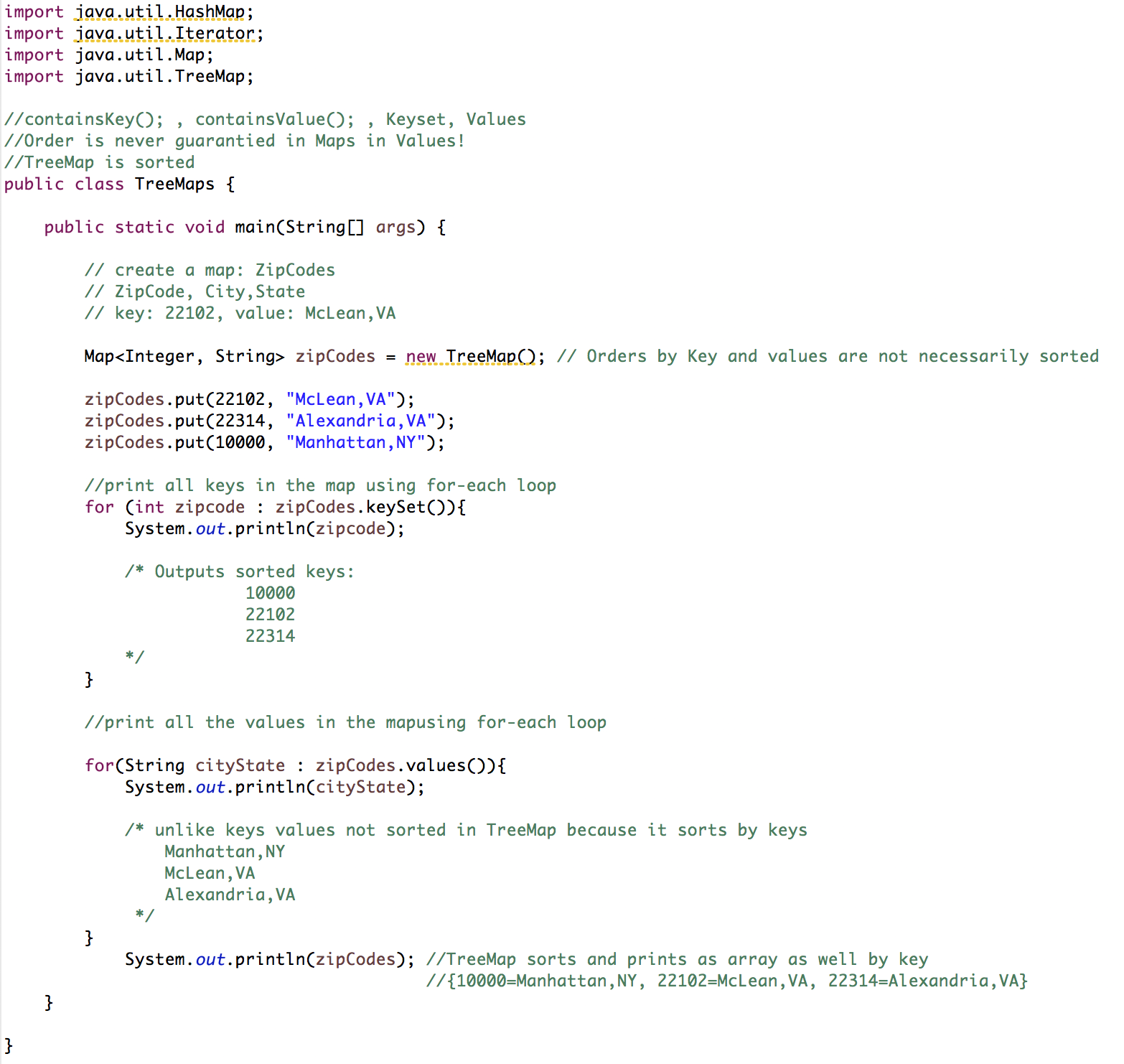
**Example:** The following example shows the usage of java.util.Collections.sort() // [(link)](https://www.tutorialspoint.com/java/util/collections_sort_comparable.htm)

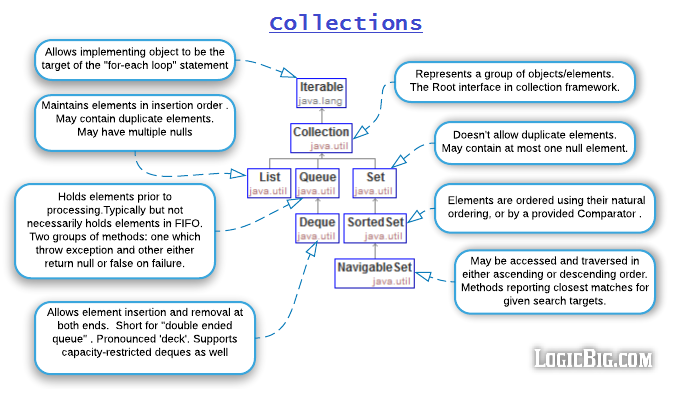
****

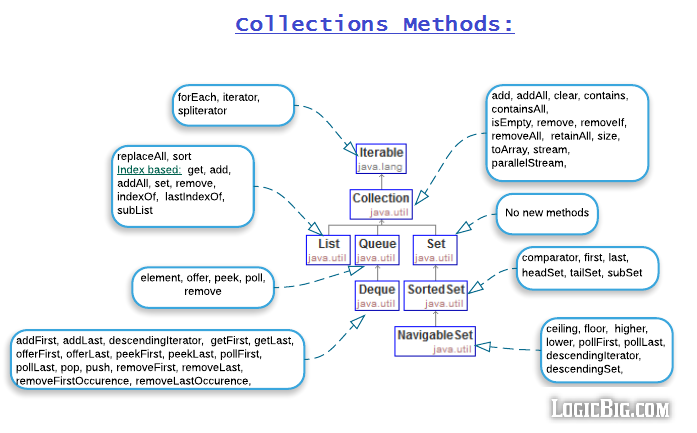
**Map Examples:**

****

**TreeMap**

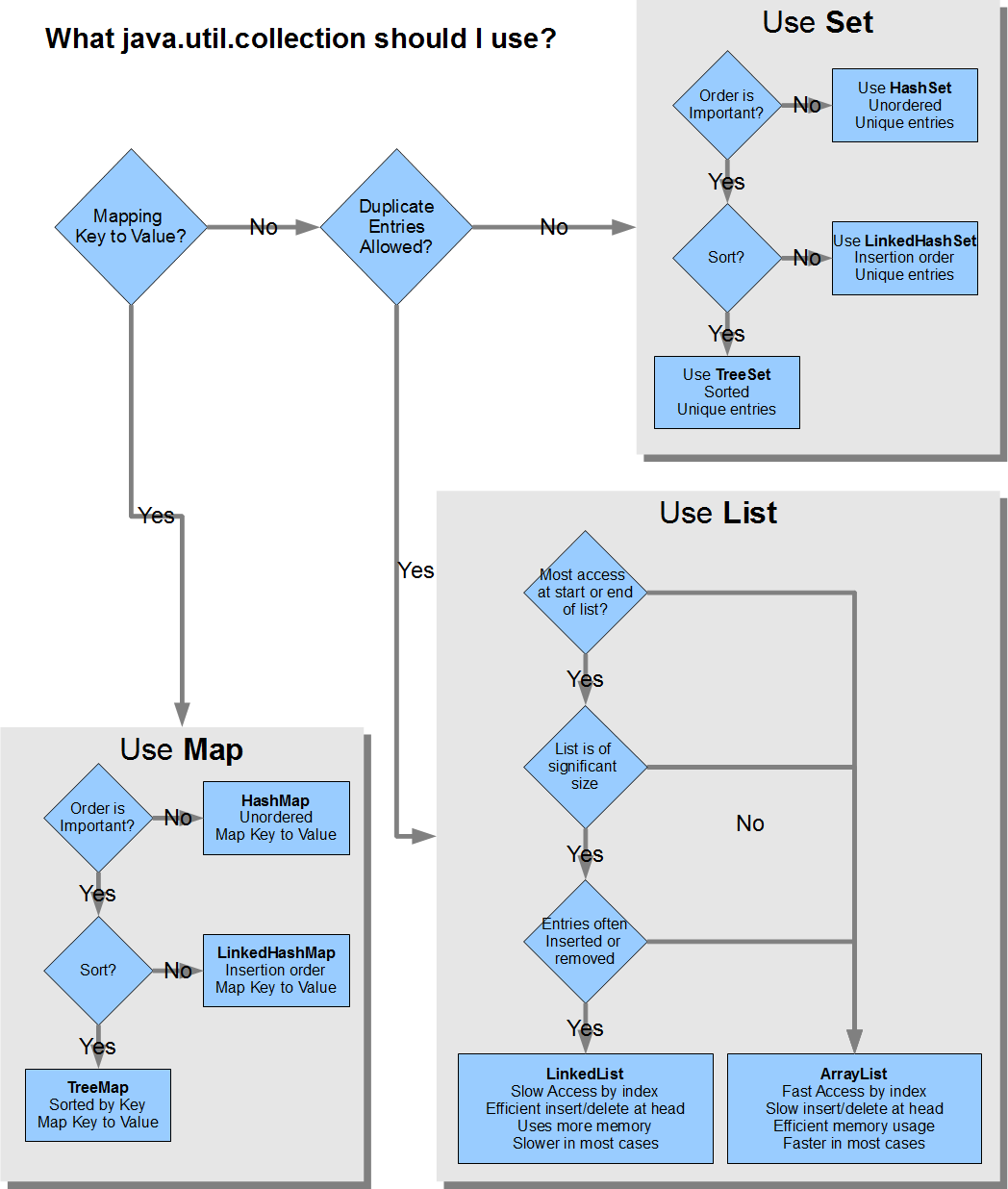
****

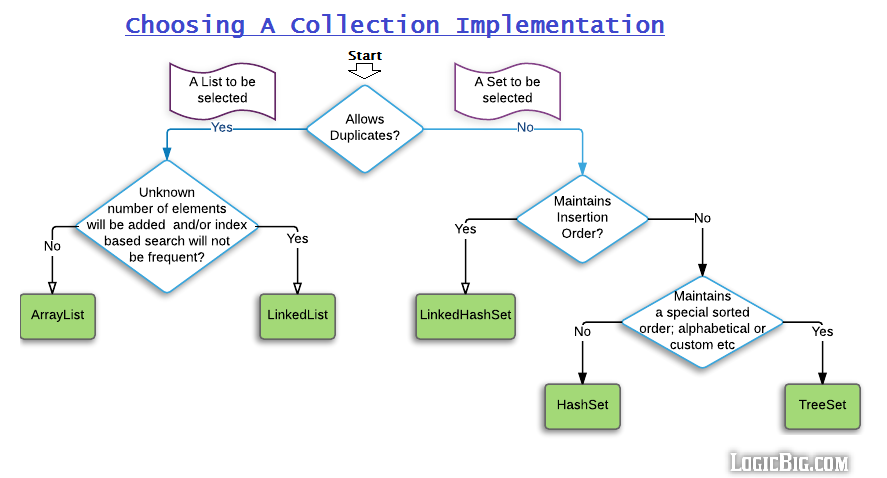




**Note**: Set doesn't have get(int index) method because no order is maintained with Set so elements don't have fixed index.

|  |  |  |  |
| --- | --- | --- | --- |
| **Impl** | [**ADT**](http://www.logicbig.com/quick-info/software-engineering/abstract-data-type/) | **Data Structure** | **Performance** |
| **ArrayList** (sync) | List | Array of objects. A new array is created and populated whenever elements are added beyond the current length (capacity) of the underlying array. | **add(E element) method:** [O(1)](http://www.logicbig.com/quick-info/software-engineering/big-o-notation/) amortized. That is, adding n elements within capacity: constant time O(1). Adding an element beyond capacity: [O(n)](http://www.logicbig.com/quick-info/software-engineering/big-o-notation/) times.  It's better to specify initial capacity at construction if known.  **remove(int index)**: O(n - index), removing last is O(1). **All other operations including get(int index)** run in linear time [O(1)](http://www.logicbig.com/quick-info/software-engineering/big-o-notation/) .  The constant factor of O(1) is low compared to that for the LinkedList implementation. |
| **LinkedList** (sync) | List, Deque | [Doubly-linked list](https://en.wikipedia.org/wiki/Doubly_linked_list). Each element has memory addresses of the previous and next item used internally. | **get(int index), remove(int index)**: O(n) **add(E element) and others**: Constant time O(1). |
| **Vector** (sync) (Legacy) | List | Array of objects. Similar to ArrayList | Similar to ArrayList but slower because of synchronization. |
| **Stack** extends Vector (sync) (Legacy) | List | Array of objects. [LIFO](https://en.wikipedia.org/wiki/Stack_(abstract_data_type)) (Last in first out). It provides addition methods **empty(), peek(), pop(), push(E e) and search(Object o)** | Similar to Vector/ArrayList but slower because of synchronisation. |
| **HashSet** (sync) | Set | Backed by HashMap (a [Hash table data structure](https://en.wikipedia.org/wiki/Hash_table)). Elements of the set are populated as key of the HashMap. Allows at most one null. | **add, remove, contains, size**: O(1) **Iteration**: O(n + capacity). Better don't set initial capacity (size of backing hasMap) too high or load factor too low if iteration is frequently used. |
| **LinkedHashSet** (sync) | Set | Backed by LinkedHashMap where elements of this LinkedHashSet are populated as key of the Map. Maintains elements in insertion order. Allows at most one null. | **add, remove, contains, size**: O(1) **Iteration**: O(n), slightly slow that of HashSet, due to maintaining the linked list. |
| **TreeSet** (sync) | NavigableSet | Backed by TreeMap (a [red-black tree data structure](https://en.wikipedia.org/wiki/Red%E2%80%93black_tree)). The elements of this set are populated as key of the Map. Doesn't permit null. | **add, remove, contains**: O(log n)  **Iteration**: O(n) slower than HashSet. |
| **EnumSet** (sync) | Set | [Bit vectors](https://en.wikipedia.org/wiki/Bit_array) All of the elements must come from a single enum type. | **All methods**: O(1). Very efficient |
| **PriorityQueue**  (sync) | Queue | [Binary Heap](https://en.wikipedia.org/wiki/Binary_heap) Unbounded Elements are ordered to their [natural ordering](https://docs.oracle.com/javase/tutorial/collections/interfaces/order.html) or by a provided Comparator. | **offer, poll, remove() and add**: O(log n)  **remove(Object), contains(Object)** O(n)  **peek, element, and size**: O(1) |
| **ArrayDeque**  (sync) | Dequeue | Resizable-array (similar to ArrayList).  Unbounded Nulls not permitted. | **remove, removeFirstOccurrence, removeLastOccurrence, contains, iterator.remove(), and the bulk operations**: O(n)  **All other operations** O(1) amortized |





**Java Collections tutorial and examples:**

<http://www.vogella.com/tutorials/JavaCollections/article.html>

<http://tutorials.jenkov.com/java-collections/list.html>

<http://www.logicbig.com/tutorials/core-java-tutorial/java-collections/java-collection-cheatsheet/>

<https://www.javatpoint.com/collections-in-java>

<https://www.tutorialspoint.com/java/java_collections.htm>

--- End of Java notes ---