***Creating and Manipulating Strings:***

* *The string class is such a fundamental class and is a sequence of characters.*

|  |
| --- |
| *String name = “fluffy”*  *String name = new String(“fluffy”);* |

* *Both give you a reference variable named “name” pointing to the String object called “fluffy”. Since String is a sequence of characters it implements interface CharSequence. The interface is a general way of representing several classes, including String and StringBuilder.*
* *Placing one String before the other String and combining them is called String Concatenation,*
* *If both operands are numeric, it does a numeric addition. If either operand is a string + refers to as concatenation.*
* *Once a String object is created, it is not allowed to change. It cannot be made larger or smaller and you cannot change one of the characters inside it.*
* *Mutable is another word for changeable. Immutable is the opposite. An object that cant be changed once its created.*
* *Even though String class is immutable it can still be used in a mutable class. You can even make the instance variable final so that the compiler reminds you if you accidentally change.*
* *Also, immutable classes in Java are final, which prevents subclass creation. You wouldn’t want a subclass adding mutable behavior.*
* *The method* ***length()*** *returns the number of characters in the String. The zero counting happens only when you are using indexes or positions within a list. When determining total size or length, Java uses normal counting again.*
* *The method* ***charAt()*** *lets you query the string to find out what character is at specific index. Since indexing starts counting with 0, charAt(0) returns the first character in the sequence. charAt(7) asks for eight character if not present, Java will throw an exception.*

|  |
| --- |
| *Syntax: char chartAt(int index)* |

* *The method* ***indexOf****() looks at the characters in the string and finds the first index that matches the desired value. Indexof can work with an individual character or a whole String as input. It can also start from a requested position. A char can be passed to an int parameter type. Unlike charAt(), the indexOf() method doesn’t throw an exception if it can’t find a match. indexOf() returns –1 when no match is found. Because indexes start with 0, the caller knows that –1 couldn’t be a valid index. This makes it a common value for a method to signify to the caller that no match is found.*

|  |
| --- |
| *int indexOf(int ch)*  *int indexOf(int ch, int fromIndex)*  *int index(String str)*  *int index(String str, int fromIndex)* |

* *The method substring() also looks for characters in the String. It returns parts of the String. The first parameter is the index to start with for the returned string. As usual, this is a zero based index. There is an optional second parameter, which is the end index you want to stop at. This is stopAt not include. This means endIndex parameter is allowed to be 1 past the end of the sequence if you want to stop at the end.*

|  |
| --- |
| *String substring(int beginIndex)*  *String substring(int beginIndex, int endIndex)* |

*The method returns the string starting from the requested index. If an end index is requested, it stops right before that index. Otherwise, it goes to the end of the string.*

* *toUpperCase(): Converts any lowercase characters to uppercase in the returned string. toLowerCase(): Converts any uppercase characters to lowercase in the returned string. These methods leave alone any characters other than letters. Also, strings are immutable, so the original string stays same.*
* *The method equals() checks whether two string objects contain exactly same characters in the same order. The equalsIgnoreCase() method checks whether two string objects contain the same characters with the exception that it will convert its characters case if needed.*

|  |
| --- |
| *Boolean equals(Object obj)*  *Boolean equalsIgnoreCase(String str)* |

*Equals() takes an object as the method is same for all objects equalIgnoreCase is specially used for strings.*

* *startsWith() and endsWith() methods look at whether the provided value matches part of the String.*

|  |
| --- |
| *Boolean startsWith(String prefix)*  *Boolean endsWith(String suffix)* |

* *The replace() method does a simple search and replace on the string. There’s a version that takes char parameters as well as version that takes CharSequence parameters.*
* *The contains() method looks for matches in the string. It isn’t as particular as startsWith() and endWith()- the match can be anywhere in the string. It is case sensitive*

|  |
| --- |
| *Boolean contains (CharSequence charSeq)* |

* *The strip() and trim() methods remove whitespaces from beginning and end of a string. The strip() method is new in Java11. It does everything the trim() does but it supports Unicode. Additionally, stripLeading() and stripTrailing() methods were added in Java11. The stripLeading() removes spaces from beginning of the string and leaves it at the end. It removes the whitespaces at the end.*

|  |
| --- |
| *String strip()*  *String stripLeading()*  *String stripTrailing()*  *String trim()* |

* *The intern() method returns the value from the string pool if it is there. Otherwise it adds the value to the string pool. String intern()*

*StringBuilder:*

* *A small program can create a lot of String objects very quickly.*
* *Example:*

|  |
| --- |
| *10. String alpha = "";*  *11: for(char current = 'a'; current <= 'z'; current++)*  *12: alpha += current;*  *13: System.out.println(alpha);*  *The empty String on line 10 is instantiated, and then line 12 appends an "a". However, because the String object is immutable, a new String object is assigned to alpha, and the "" object becomes*  *eligible for garbage collection. The next time through the loop, alpha is assigned a new String object, "ab", and the "a" object becomes eligible for garbage collection. The next iteration assigns alpha to "abc", and the "ab" object becomes eligible for garbage collection and so on.* |

* *This sequence of events continues and after 26 iterations through the loop a total of 27 objects are instantiated most of which are immediately eligible for garbage collection.*
* *This is very inefficient. Luckily, Java has a solution. The StringBuilder class creates a String without storing all those interim String values. Unlike the String class, StringBuilder is immutable.*
* *StringBuilder every time in for loop keeps appending the value to the string. This code reuses the same StringBuilder without creating an interim String each time.*
* *Sslower than StringBuilder.*
* *Chaining with String was a new String. However, chaining with StringBuilder changes its own state and returns a reference to itself.*

*Creating a StringBuilder:*

* *Three ways to construct a StringBuilder:*

|  |
| --- |
| *StringBuilder sb1 = new StringBuilder();*  *StringBuilder sb2 = new StringBuilder(“animal”);*  *StringBuilder sb3 = new StringBuilder(10);* |

* *The first says to create a StringBuilder containing an empty sequence of characters and assign sb1 to point to it. The second says to create a StringBuilder containing a specific value and assign sb2 to point to it. The third tells Java that we have some idea on how big the eventual value would be and would like the StringBuilder to reserve a certain capacity or number of slots for characters.*

*StringBuilder Methods:*

* *charAt(), indexOf(), length(), substring() works same as it works for String. Substring() would return a string instead of stringbuilder. It is a method that inquires about what the state of the StringBuilder happens to be.*
* *The method append() is most frequently used in StringBuilder. It adds the parameter to the StringBuilder and returns a reference to the current StringBuilder.*

|  |
| --- |
| *StringBuilder append(String str)* |

*Append() is called directly after the constructor. By having all these method signatures, you can just call append() without having to convert your parameter to a String first.*

* *The insert() method adds characters to the StringBuilder at the requested index and returns a reference to the current StringBuilder.*

|  |
| --- |
| *StringBuilder insert(int offset, String str)* |

*Offset is the index where you want to insert the requested parameter.*

* *The delete() method is the opposite of the insert() method. It removes the characters from the sequence and returns a reference to the current StringBuilder. The deleteCharAt() method is convenient when you want to delete only one character.*

|  |
| --- |
| *StringBuilder delete(int startIndex, int endIndex)*  *StringBuilder deleteCharAt(int index)* |

*The delete() method is more flexible than some others when it comes to array indexes. If you specify a second parameter that is past the end of the StringBuilder.*

* *The replace() method works differently for StringBuilder than it did for String.*

|  |
| --- |
| *StringBuilder replace(int startIndex, int endIndex, String newString)*  *Eg: StringBuilder builder = new StringBuilder(“pigeon dirty”);*  *Builder. Replace(3,6,”sty”)*  *System.out.println(builder); //pigsty dirty* |

*First, Java deletes the characters starting with index 3 and ending with index 6. This gives us pig dirty. Then Java inserts to the value “sty” in that position.*

* *The method reverse() reverses all the characters in the sequence and returns a reference to the current StringBuilder.*

|  |
| --- |
| *StringBuilder reverse()* |

* *The toString() method converts a StringBuilder into a String. String toString()*

*Often, StringBuilder is used internally for performance purposes but the end result needs to be a String.*

*Understanding Equality:*

* *Always see if the references are referring to the same object.*

*Eg: StringBuilder one = new StringBuilder();*

*StringBuilder two = new StringBuilder();*

*StringBuilder three = one.append(“a”);*

*System.out.println(one==two); //false*

*System.out.println(one == three) //true*

*One and two are completely separate StringBuilder objects giving us two objects. One and three both point to the same object hence second print returns true*

* *The authors of String class implemented a standard method called equals to check the values inside the String rather than the String reference itself. If a class doesn’t have an equals method, java determines whether the references point to the same object which is what == does. For this to workout in StringBuilder, you can call toString() on StringBuilder to get a string checked for equality. If you call equals() on two StringBuilder instances it will check for reference equality.*
* *Eg:*

*String string = “a”;*

*StringBuilder builder = new StringBuilder(“a”);*

*System.out.println(string == builder) //Doesn’t compile*

*== is checking for object reference equality. The compiler is smart enough to know that the two references cant possibly point to the same object when they are completely different types.*

*String Pool:*

* *Since strings are everywhere they use up a lot of memory. In some production applications, they can use a large amount of memory in the entire program. Java realizes that many strings repeat in the program and solves this issue by reusing common ones.*
* *The string pool is commonly known as intern pool is a location in the JVM that collects all these strings. The string pool contains literal values and constants that appear in the program.*
* *String x = “Hello World”; String y = new String(“Hello World”); System.out.println(x == y);*

*The former says to use the string pool normally. The second says JVM to create a new object even though it is less efficient. The intern() method will use an object in the string pool if one is present. If the literal isn’t there in string pool Java will add it this time.*

*String name =”Hello World”;*

*String name2 = new String(“Hello World”).intern();*

*System.out.println(name == name2);*

*First, we tell Java to use the String pool normally for name. then for name2, we tell java to create a new object using the constructor but to intern it and use the string pool anyway. Since both the variables point to the same reference in the string pool, we can use the == operator.*

* *Never use the intern() or == to compare String objects in your code.*

*Understanding Java Arrays:*

* *An array is an area of memory on the heap with space for a designated number of elements.*
* *A string is implemented as an array with some methods that you might want to use when dealing with characters specifically.*
* *A StringBuilder is implemented as an array where the array object is replaced with a new bigger array object when it runs out of space to store all the characters.*

*Creating an array of Primitives:*

* *Int numbers[] = new int[3]; //only size of 3*
* *Int numbers2[] = new int[] {42, 55, 99};// size of 3 along with initial values*

*Java finds this expression as redundant. Since you are specifying type of array on the ;eft side, Java already knows the type. And since you are specifying the initial values, it already knows its size. Int[] numbers2 = {42, 55, 99};*

*This approach is called anonymous array. It is anonymous because you don’t specify type and size.*

* *We can call equals() because an array is an object. It returns true because of reference equality. The equals() method on array doesn’t look at the elements of an array. When you print a string it returns [Ljava.lang.String; @160bc7c0. L here means an array, java.lang.String is the reference type, 160bc7c0 is the hash code.*
* *Since Java5, java has provided a method that prints an array nicely. Arrays.toString(bugs) would print a complete array. The array doesn’t allocate space for string objects. Instead it allocates space for a reference to where the objects are really stored.*

|  |
| --- |
| *String[] strings = { “stringValue” }*  *Object[] objects = strings;*  *String[] againStrings = (String[]) objects;*  *againStrings[0] = new StringBuilder(); //Doesn’t compile*  *objects[0] = new StringBuilder();* |

* *First line creates an array of String. Second line doesn’t require a cast because Object is a broader type than String. Third line, cast is needed because we are moving to a more specific type. 4th line doesn’t compile because a String[] only allows String objects and StringBuilder is not a string.*
* *Last line – From the point of compiler, this is just fine. A StringBuilder object can clearly go in to an Object[]. The problem is that we don’t actually have an Object[].*
* *We have a String[] referred to from an Object[] variable. At runtime, the code throws an ArrayStoreException.*

*Sorting in Arrays:*

* *Java makes it easy to sort an array by providing a sort method or rather a bunch of sort methods. Just like StringBuilder allowed you to pass almost anything to append() you can pass almost any array to Arrays.sort().*
* *Arrays is the first class provided by Java we have used that needs an import.*

*Searching:*

* *Java provides a convenient way to search – but only if the array is actually sorted.*
* *Target element found in array – Return Index of match*
* *Target element not found in array – Negative value showing one smaller than the negative of the index, where a match needs to be inserted to preserve sort order.*
* *Unsorted array – A surprise result is unpredictable*

|  |
| --- |
| *Example:*  *Int[] numbers = {2,4,6,8}*  *System.out.println(Arrays.binarySearch(numbers, 2) // 0*  *System.out.println(Arrays.binarySearch(numbers, 4 // 1*  *System.out.println(Arrays.binarySearch(numbers, 1) // -1*  *System.out.println(Arrays.binarySearch(numbers, 3) // -2*  *System.out.println(Arrays.binarySearch(numbers, 9) // -5*  *1 isnt in the list, the search can determine that it should be inserted at index 0 to preserve the sorted order. Since 0 means already something for index, Java needs to subtract 1 to give us answer of -1. Similarly for 3 and 5* |

* *The binary search spilts an array into two equal pieces and determines which half the target is in. it repeats this process until one element is left in.*

*Comparing:*

* *Java also provides methods to compare two arrays to determine which is smaller.*

***Compare():***

* *A negative number means that the first array is smaller than the second.*
* *A zero means that the arrays are equal.*
* *A positive number means that the first array is larger than the second.*
* *If both the arrays are the same length and have the same values in each spot in the same order return zero.*
* *If all the elements are the same but the second array has extra elements at the end, return a negative number.*
* *If all the elements are the same but the first array has extra elements at the end, return a positive number.*
* *If the first element that differs is smaller in first array, return a negative number.*
* *If the first element that differs is larger in first array, return a positive number.*
* *Null is smaller than any other value.*
* *For numbers, normal numeric order applies.*
* *For strings, one is smaller if it is a prefix of another.*
* *For strings/characters, numbers are smaller than letters*
* *For strings/characters, uppercase is smaller than lowercase.*

*Arrays.compare():*

|  |  |  |
| --- | --- | --- |
| *New int[]{1,2}* | *New int[] {1}* | *Returns a positive value. The first element is the same but the first array is longer* |
| *New int[]{1,2}* | *New int[]{1,2}* | *Zero – Exact match* |
| *New String[]{“a”}* | *New String[]{“aa”}* | *Negative number – the first element is the substring of the second.* |
| *New String[]{“a”}* | *New String[]{“A”}* | *Positive number – Uppercase is smaller than lowercase* |
| *New String[]{“a”}* | *New String[]{null}* | *Positive number- null is smaller than letter* |

* *While comparing two arrays, the type must be same else the code doesn’t compile.*

***Mismatch():***

* *If the arrays are equal, mismatch() returns -1. Otherwise it will return the first index where they differ.*

|  |
| --- |
| *System.out.println(Arrays.mismatch(new int[]{1}, new int[]{1});*  *System.out.println(Arrays.mismatch(new String[]{“a”}, new String[]{“A”}));*  *System.out.println(Arrays.mismatch(new int[]{1,2}, new int[]{1}));*  *First one, arrays are same so output is -1*  *Second one, 0 index are not equal so output is 0*  *Third one, first element is equal but second element there is no entry so output is 1* |

*Varargs:*

* *public static void main(String[] args)*
* *public static void main(String args[])*
* *public static void main(String… args) //varargs*

*Understanding an ArrayList:*

* *An array has one shortcoming. You have to know how many elements will be in an array when you create it. Just like StringBuilder, an ArrayList can change capacity at runtime as needed.*
* *Like an array, an ArrayList is an ordered sequence that allows duplicates. As Arrays.sort, ArrayList requires an import. (import java.util.\*, java.util.ArrayList)*
* *Three ways to create an ArrayList:*

|  |
| --- |
| *ArrayList list1 = new ArrayList();*  *ArrayList list2 = new ArrayList(10);*  *ArrayList list3 = new ArrayList(list2);*  *ArrayList<String> list4 = new ArrayList<String>();*  *ArrayList<String> list5 = new ArrayList<>();* |

* *The first says to create an ArrayList containing space for the default number of elements but no to fill any slots yet. The second says to create an ArrayList containing a specific number of slots but again not to assign any. The final one tells Java to make another copy of list. We copy both the size and contents of that ArrayList.*
* *Java5 allows you to tell the compiler what the type would be in between angular brackets. Starting with Java7, you can even omit that type from right end having angular brakcets.*

*Using var with ArrayList:*

* *Now that var can be used to obscure data types, there is a whole new group of questions that can be asked with generics.*

|  |
| --- |
| 1. *Var strings = new ArrayList<String>();*   *Strings.add(“a”);*  *For(String s: strings){}*   1. *var list = new ArrayList<>();* 2. *var list = new ArrayList<>()*   *list.add(“a”);*  *for(String s: list) {} //Doesn’t compile* |

* *The type of var is ArrayList<String>. This means you can add a String or loop through the String objects.*
* *The second one does compile. The type of var here is ArrayList<Object>. Since there isn’t a type specified for the generic, Java has to assume the ultimate superclass.*
* *For the third one, the type of var is ArrayList<Object>. Since there isn’t a type in the diamond operator, Java has to assume the most generic option it can. Therefore, it picks Object, the ultimate superclass. Adding a String to the list is fine. You can add any subclass of object. However, in the loop, we need to use the Object type rather than String.*
* *ArrayList implements an interface called List. List is an interface and it cant be instantiated.*

|  |
| --- |
| *List<String> list6 = new ArrayList();*  *ArrayList<String> list7 = new List<>(); //Doesn’t compile* |

* *ArrayList also implements toString(), so you can easily see the contents just by printing it.*

*Methods:*

* *The add() method insert a new value in the ArrayList.*

|  |
| --- |
| *Boolean add(E element)*  *Void add(int index, E element)* |

* *The remove() method removes the first matching value in the ArrayList or remove the element at the specified index.*

|  |
| --- |
| *Boolean remove(Object object)*  *E remove(int index)* |

*The Boolean returns true if the match is removed.*

* *The set() method changes one of the elements of the ArrayList without changing the size.*

|  |
| --- |
| *E set(int index, E newElement)* |

*The E return type is the element that got replaced.*

* *The isEmpty() and size() methods look at how many of the slots are in use.*
* *The clear() method provides an easy way to discard all elements of the ArrayList. Void clear()*
* *The contains() method checks whether a certain value is in the ArrayList.*

*Boolean contains(Object object)*

*This method calls equals() on each element of the ArrayList to see whether there are any matches. Since String implements equals() this works out well.*

* *Finally, ArrayList has a custom implementation of equals() so you can compare two lists to see whether they contain the same elements in the same order.*

*Boolean equals(Object object)*

*Wrapper Classes:*

* *Each primitive type has a wrapper class, which is an object type that corresponds to the primitive.*

|  |  |  |
| --- | --- | --- |
| *Primitive Type* | *Wrapper Class* | *Example of Creating* |
| *boolean* | *Boolean* | *Boolean.valueOf(true)* |
| *byte* | *Byte* | *Byte.valueOf((byte)1)* |
| *short* | *Short* | *Short.valueOf((short)1)* |
| *long* | *Long* | *Long.valueOf(1)* |
| *int* | *Integer* | *Integer.valueOf(1)* |
| *float* | *Float* | *Float.valueOf(float)1.0)* |
| *double* | *Double* | *Double.valueOf(1.0)* |
| *char* | *Character* | *Character.valueOf(‘c’)* |

* *Each wrapper class also has a constructor. It works the same way as valueOf() but isn’t recommended for new code. The valueOf() allows object caching. The wrapper classes are immutable and take advantage of same caching as String does when its value is same using String pool.*
* *The wrapper classes also have a method that converts back to a primitive. You don’t need to know much about the valueOf() or intValue() because the autoboxing has removed the need for them.*
* *There are also methods for converting a String to a primitive or wrapper class. The parse methods such as parseInt() return a primitive, and the valueOf() method returns a wrapper class*

|  |  |  |
| --- | --- | --- |
| *Wrapper class* | *Converting String to Primitive* | *Converting String to Wrapper class* |
| *Boolean* | *Boolean.parseBoolean(“true”)* | *Boolean.valueOf(“TRUE”)* |
| *Byte* | *Byte.parseByte(‘1”)* | *Byte.valueof(“2”)* |
| *Short* | *Short.parseShort(“1”)* | *Short.valueOf(“2”)* |
| *Integer* | *Integer.parseInt(“1”)* | *Integer.valueOf(“2”)* |
| *Long* | *Long.parseLong(“1”)* | *Long.valueOf("2")* |
| *Float* | *Float.parseFloat("1")* | *Float.valueOf("2.2")* |
| *Double* | *Double.parseDouble("1”)* | *Double.valueOf("2.2")* |
| *Character* | *None* | *None* |

* *One advantage of a wrapper class over a primitive is that because it’s an object, it can be used to store a null value. While null values aren’t particularly useful for numeric calculations, they are quite useful in data based services.*

*Autoboxing and Unboxing:*

* *Since Java5, you can just type the primitive value, and Java will convert it to the relevant wrapper class for you. This is called* ***Autoboxing.***
* *The reverse conversion of the wrapper class to primitive value is called* ***Unboxing.***

|  |
| --- |
| *List<Integer> weights = new ArrayList<>();*  *Integer w = 50;*  *weights.add(w); //50*  *weights.add(Integer.valueOf(60)); //[50,60]*  *weights.remove(new Integer(50)); //[60]*  *double first = weights.get(0); //60.0* |

* *Second line autoboxes the int primitive to an Integer object.*
* *Line 3 adds it to a list. Line 4 shows that you can still write code the long way and pass in the wrapper object. Last line retrieves the first integer in the list unboxes it as a primitive and implicitly casts it to double.*

|  |
| --- |
| *List<Integer> heights = new ArrayList<>();*  *heights.add(null);*  *int h = heights.get(0); //NullPointerException* |

*We added a null to a list. This is legal because a null reference can be assigned to any reference variable. Next, we try to unbox that null to an int primitive. This is a problem. Java tries to get the int value of null. Since calling any method on null gives a NullPointerException, but that is just what we get.*

|  |
| --- |
| *List<Integer> numbers = new ArrayList<>();*  *numbers.add(1);*  *numbers.add(2);*  *numbers.remove(1);*  *System.out.println(numbers);* |

*It actually outputs [1]. After adding these two values, the list contains [1,2]. We then request the element with index 1 be removed. There is a remove method that takes an int parameter. Java calls that method instead of autoboxing. If you want to remove 1 you can write it as numbers.remove(new Integer(1)) to force wrapper class use.*

*Converting between Array and list:*

|  |
| --- |
| *List<String> list = new ArrayList<>();*  *list.add(“hawk”);*  *list.add(“robin”);*  *Object[] objectArray = list.toArray();*  *String[] stringArray = list.toArray(new String[0]);*  *list.clear();*  *System.out.println(objectArray.length); //2*  *System.out.println(stringArray.length); //2* |

*ArrayList knows how to convert itself to an array. The only problem is that it defaults to an array of class Object. The advantage of specifying a size of 0 for the parameter is that java will create a new array of the proper size for the return value. If the ArrayList fits in that array, it will be returned. Otherwise, a new one will be created.*

* *Converting from an array to a List is more interesting. We will show you two methods to do this conversion. One option is to create a list that is linked to the original array. When a change is made to one, it is available in the other. It is a fixed size list and is also known as a backedList because the array changes with it.*

|  |
| --- |
| *String[] array = {“hawk”, “robin”}; // [hawk, robin]*  *List<String> list = Arrays.asList(array); //Fixed size list*  *System.out.println(list.size()); //2*  *list.set(1,”test”); // [hawk, test]*  *array[0] = “new”; // [new, test]*  *System.out.println(Arrays.toString(array)); //[new, test]*  *list.remove(1); //throws UnsupportedOperationException* |

* *Converts an array to a list. Note that this isn’t the java.util.ArrayList we have grown to. It is a fixed size backed version of a List. set() merely replaces an existing value. It updates both array and list because they point to the same data store. Last line throws an exception because we aren’t allowed to change the size of the list.*
* *Another option is to create an immutable list. That means you cant change the values or size of the list. You can change the original array, but changes will not be reflected in the immutable list.*

|  |
| --- |
| *String[] array = {“hawk”,”robin”} // [hawk, robin]*  *List<String> list = list.of(array); //Immutable list*  *System.out.println(list.size()); //2*  *Array[0]= “new”;*  *System.out.println(Arrays.toString(array));*  *System.out.println(list);*  *List.set(1,”test”); //throws unsupported exception* |

*Creates an immutable List. It contains two values that array happened to contain at the time the list was created. There is a change to an array but list still holds the original values. This is because it is an immutable copy of the original array. Last line is changing a list value in an immutable list isn’t allowed.*

***Using Varargs to create a list:***

* *Using varargs allows you to create a List in cool way:*

*List<String> list1 = Arrays.asList(“one”,”two”);*

*List<String> list2 = List.of(“one”,”two”);*

*Both of these methods take varargs, which let you pass in an array or just type out the string values. This is handy in testing because you can easily create and populate a List on one line. Both methods create fixed size arrays. If you will need to later add or remove elements, you will still need to create an ArrayList using the constructor.*

|  |  |  |  |
| --- | --- | --- | --- |
|  | *toArray()* | *Arrays.asList()* | *List.of()* |
| *Type converting from* | *List* | *Arrays or varargs* | *Array or varargs* |
| *Type Created* | *Array* | *List* | *List* |
| *Allowed to remove values from created object* | *No* | *No* | *No* |
| *Allowed to change values in the created object* | *Yes* | *Yes* | *No* |
| *Changing values in the created object affects the original or vice versa* | *No* | *Yes* | *N/A* |

* *Notice that none of the options allow you to change the number of elements. If you want to do that, you will need to actually write logic for creating new object.*

*List<String> fixedSizeList = Arrays.asList(“a”,”b”,”c”);*

*List<String> expandableList = new ArrayList<>(fixedSizeList);*

***Sorting:***

* *Sorting an ArrayList is similar to sorting an array. You just need to use a different helper class*

*List<Integer> numbers = new ArrayList<>();*

*numbers.add(99);*

*numbers.add(9);*

*numbers.add(55);*

*Collections.sort(numbers);*

*System.out.println(numbers); // [9,55,99]*

*Introducing Sets:*

* *A set is a collection of objects that cannot contain duplicates. If you try to add a duplicate to a set, the API will not fulfil the request. All the methods you learned from an ArrayList apply to Set with the exception of those taking an index as a parameter.*
* *A Set isn’t ordered, so it wouldn’t make sense to talk about first element. This means you cannot call set(index, value) or remove(index).*
* *You can call other methods like add(value) or remove(value). When trying to add a duplicate value the method returns false and doesn’t add the value. HashSet and TreeSet are two different implementations. TreeSet is used when sorting is specific.*

|  |
| --- |
| *Set<Integer> set = new HashSet<>();*  *System.out.println(set.add(66)); //true*  *System.out.println(set.add(66)); //false*  *System.out.println(set.size()); //1*  *Set.remove(66):*  *System.out.println(set.isEmpty()); //true* |

*Creates a new Set that declares only unique elements are allowed. Hence only one is allowed to be added.*

*Maps:*

* *A map is used to identify values. The most common implementation of Map is HashMap. Some of the methods are same as those in ArrayList like clear(), isEmpty(), size().*

|  |  |
| --- | --- |
| *Method* | *Description* |
| *V get(Object key)* | *Returns the value mapped by key or null if none is mapped* |
| *V getOrDefault(object key, V other)* | *Returns the value mapped by key or other if none’s mapped* |
| *V put(K key, V value)* | *Adds or replaces key/value pair. Returns previous value or null* |
| *V remove(Object key)* | *Removes and returns value mapped to key. Returns null if none* |
| *boolean containsKey(Object key)* | *Returns whether key is in map* |
| *boolean containsValue(object value)* | *Returns whether value is in map* |
| *Set<K> keyset()* | *Returns set of all keys* |
| *Collection<V> values()* | *Returns collection of all values* |

* *Map<String, String> map = new HashMap<>();*

*map.put(“koala”,”bamboo”);*

*String food = map.get(“koala”) ; //bamboo*

*String other = map.getOrDefault(“ant”,”leaf”); //leaf*

*for(String key: map.keySet()){*

*System.out.println(key+ “ “+ map.get(key)); //koala bamboo*

***Calculating with Math API’s:***

* *The min() and max() methods compare two values and return one of them.*

|  |
| --- |
| *Double min(double a, double b)*  *Float min(float a, float b)*  *Int min(int a, int b)*  *Long min(long a, long b)* |

* *The round() method gets rid of the decimal portion of the value choosing the higher number if appropriate.*

|  |
| --- |
| *Long round(double num)*  *Int round(float num)* |

* *The pow() method handles exponents. (double pow(double number, double exponent)*
* *The random() method returns a value greater than or equal to 0 and less than 1.*