* Generics
  + Using type as a parameter
  + Writing code that can handle many different types of objects
  + We use angle brackets <> to denote that something is a generic
  + Allows us to ensure type safety
    - We use a placeholder instead of an explicit type (T, E, ?, M, etc.)
  + Adds stability and reusability to your code.
    - At compile time, checks whether that type of placeholder is consistently used throughout your code.
  + If we wanted code that could handle any type of object parameter (without generics), we would have to pass an Object object in as a parameter.
* Without Generics

|  |
| --- |
| Person p = new Person("Bob");  Person q = (Person) someMethod(p);  //ClassCastException -> no type safety at compile time  //Opens up code to lots of potential problems   public Object someMethod(Object o){   //Checks whether has a "name" field and changes it Dog something = new Dog("Fido"); return something; //Works if something is an Object |

* + With Generics

|  |
| --- |
| Person p = new Person("Bob);  Person q = someMethod(p);   //T is the placeholder for the returned type   //You set the placeholder type when you invoke the method  //Second T (outside of <>) is the return type. Does not have to return  //type T. Generally single letter, uppercase letters  public <T> T someMethod (T something) {  //do something  //return object of type T  } |

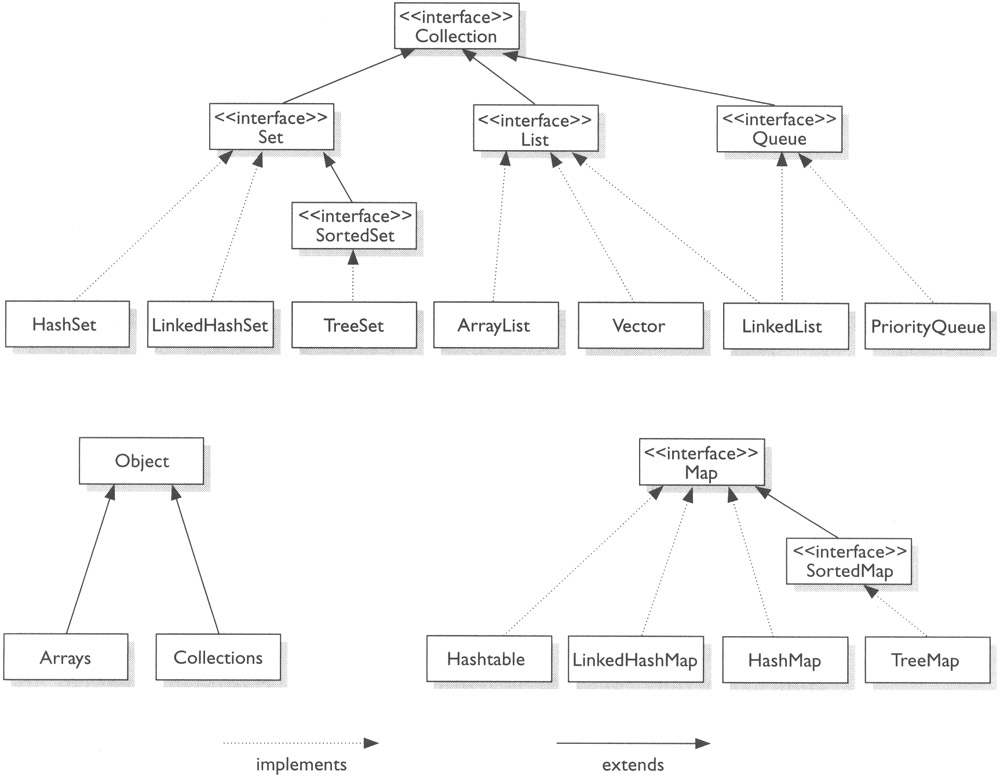
***The single, most important example of generics? Collections!***

The 3 c’s

Ok, let's start off by defining some key terms which are all going to sound the exact same: **collection** (lowercase 'C'), **Collection** (uppercase 'C'), and **Collections** (plural and uppercase 'C').

* collection
  + refers to the concept of aggregating some related data items
* Collection
  + refers to the Collection interface, defined in the java.util package
  + https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html
* Collections
  + refers to the Collections class, also defined in the java.util package
  + consists exclusively of static methods that operate on or return collections
    - .sort()
    - .swap()
    - .shuffle()
    - .add()
    - .addAl()
      * addAll(List<T>, …<T>) - adds all elements from whatever you’re iterating through to your Collection. Uses varargs
    - etc.l
  + <https://docs.oracle.com/javase/8/docs/api/?java/util/Collections.html>

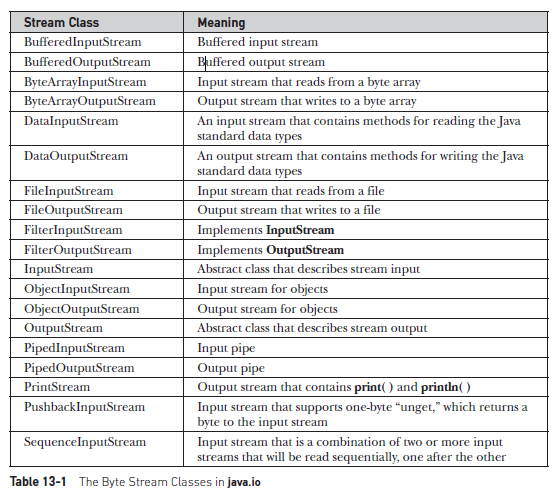
Collections Framework Diagram



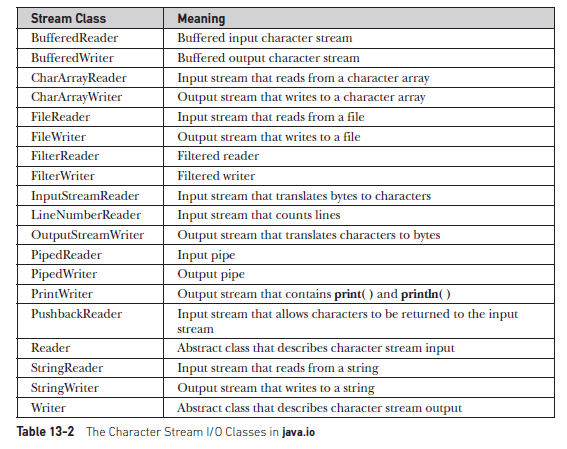
* + List - Interface
  + ArrayList - Class
    - Initial capacity 10
    - Increases automatically as elements added
    - Copies elements into new memory space once exceeds 75
      * add()
      * remove(int index)
      * size()
      * set(int index, E value)
      * isEmpty()
      * ensureCapacity(int capacity)
  + Vector - Class
    - List that dynamically increases and decreases at runtime as needed.
    - Failfast type of array - automatically tells you something is wrong
    - Good at bug detecting
    - Similar to ArrayList - dynamic array whose capacity you can set. Once reached, allocates more memory automatically.
      * trim() to set memory capacity to minimum possible
      * Method to remove specific element
  + LinkedList - Class
    - Implements List
    - Indexed but does not support random access - uses “next” feature.
    - Singly or Doubly linked lists
    - Very easy to add an element into the LinkedList by changing the references held by next
    - Downside: takes longer to retrieve specific elements; have to traverse linked list one by one
    - Not thread safe
      * add() - adds to the end of the list by default; can specify where
      * remove() - removes the head by default; can specify where
      * clear()
      * size()
      * set(index, E)
  + PriorityQueue - Class
    - Implements Queue, Collection, Iterable and Serializable
    - An unbounded priority queue based on a priority heap.
    - Elements are ordered according to their natural ordering, which is the ordering provided by the JVM. You can also order it using Comparator. The head is the least element.
    - Common methods
      * add(E e) - inserts element into the priority queue
      * offer(E e) - immediately inserts element into the queue as long as it does not violate capacity restrictions. Returns false if cannot insert, whereas add will throw an IllegalStateException.
      * clear() - clear all elements from this priority queue
      * contains(Object o) - returns true if queue contains element
      * poll() - retrieves and removes head of queue
      * peek() - retrieves but does not remove head of queue
      * remove(Object o) - removes a single instance of specified element if present
      * size() - returns the number of elements
      * toArray() - returns Object array containing all elements in the queue.
  + LinkedHashSet - Class
    - Combination of LinkedList, HashTable and Set
    - Uses a hash table to check whether an element is contained
    - Has a spliterator
    - Utilizes linked lists in each hash
      * add()
      * contains()
      * remove()
      * size()
      * isEmpty()
  + HashSet - Class
    - Similar to LinkedHashSet
    - Does not ensure order, as opposed to LinkedHashSet
    - Contains a HashMap that contains the hash mapping for the hash set
    - Has a spliterator
    - LoadFactor attribute - decimal that reflects what percentage of the capacity can be used before it doubles the capacity. Can be changed in the constructor. Can’t be changed once constructed
    - Not thread-safe
  + TreeSet - Class
    - Access and find storage information quickly
    - Does not allow duplicates
    - Not thread-safe
    - Not indexed
    - Sorted
    - Can give it a Comparator to specify ordering or use the natural ordering
  + HashMap - Class
    - Hash table that implements the Map interface
    - Hash table unsynchronized, HashMap permits nulls
    - Values contained based on the key, and you need the key to access each value.
    - Values are not in the order you place them in
      * size() returns number of mappings
      * remove()
      * replace()
      * all() add all values from another map to current hash map
      * get(key) returns object
      * isEmpty
  + HashTable - Class
    - Hash table is thread-safe implementation of hash map
      * Use ConcurrentHashMap
    - Multithread app where you don’t need to access things really quickly
    - Similar methods to HashMap
    - The keys are a Collection
  + TreeMap - Class
    - Another Map
    - Sorted by natural ordering, can use Comparator
    - HashMap vs TreeMap
      * TreeMap sorted by ascending
      * HashMap unsorted
    - Unsynchronized, not thread-safe
    - All elements have to be comparable
    - When to use
      * Your map is a dictionary and you want to print things in sorted order
      * Tight on memory and don’t want HashMap to keep expanding -> lots of hash codes end up in the same bucket
    - Lots of the same Methods
      * lowerKey - searches for lowest item
      * Clear
      * containsKey
      * keySet
* Map is not a descendant of Collection
* All of these implement the Iterable interface
  + Only iterables can be used in for-each loops.
  + Interface that, when implemented, allows the implementing type to be the subject of a for-each loop
  + Requires implementation of an Iterator
    - Every collection has an Iterator
    - Allows traversal between elements and safe removal of elements in-place
    - What does the iterator look like?

|  |
| --- |
| public interface Iterator<E>{  //Iterator over a generic  boolean hasNext();  E next();  void remove(); //Only way to safely remove element during  //traversal  }     public boolean condition (Object o) {  //Check thing  //return boolean value  }   //Filter a collection based on some condition  public void filter (Collection<?> c) {  Iterator<?> it = c.iterator(); //retrieve the iterator from c  while (it.hasNext()) {  if (!condition(it.next())) {  it.remove();  }  }  } |

* + - * Can traverse multiple iterables (non-Collection collection) in parallel (not in the threading sense). Thread-safe if the collections we’re using are thread-safe.
        + while (it.hasNext() && it2.hasNext())
  + Working with collections, lists and generic types
* **Reflections**
  + It breaks encapsulation
  + Methods for runtime inspection of objects
    - You can even look at private memory
  + It can access the:
    - Class of an object
    - Fields
    - Methods
    - Constructors
  + Within your source code, you want to protect your endpoints. Make all users authenticated. It is possible to find out everything about an object using the Reflections API. Can’t rely on making a field private if it contains sensitive information, in this case.
  + Start to think like a framework
  + It also lets you modify and instantiate things. Not just looking.
  + Can check whether an object is an instanceof a specific class (is a certain type)
  + Can check the superclass and implemented interfaces
  + You can even invoke methods
  + **com.revature.reflection**
* **Comparisons**
  + Wouldn’t it be useful to sort and compare objects based on a configurable “natural ordering”?
  + Method 1
    - Have a class implement the Comparable interface
      * has 1 method- compareTo(T,O)
      * Comparable is a Functional Interface
    - If a class implements Comparable, it MUST implement a compareTo(...) method.
    - <https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html>
  + Method 2
    - Create an external class which implements a different interface called Comparator.
      * Comparator is a Functional Interface
    - Override the compare() method.
    - Needed for the Collections sort() method
  + Comparable vs Comparator
    - https://www.geeksforgeeks.org/comparable-vs-comparator-in-java/
  + **com.revature.compare**
* Functional Interface
  + - Interface w/ single abstract method
  + <https://www.oreilly.com/learning/java-8-functional-interfaces>
  + Good for Lambdas!
* Lambdas
  + Provides a clear and concise way to represent one method interface using an expression.
  + Improves the Collection libraries making it easier to iterate through, filter, and extract data from a Collection.
  + New concurrency features improve performance in multicore environments.
  + How to Lambda: <https://www.geeksforgeeks.org/lambda-expressions-java-8/>
  + Predicate chaining: <https://www.geeksforgeeks.org/java-8-predicate-with-examples/>
    - Use “and” to chain predicate names together.
  + Can implement and use Functional Interfaces on-the-fly
* **Scanner**
  + Reading from system.in
* Serializing
  + converting a Serializable Java object to a byte stream.
    - ObjectInputStream - for serializing
    - ObjectOutputStream - for serializing
    - Serialized ID
  + Transient - keyword that marks a variable that will not be serialized
  + If there’s a field we don’t want to make Serializable, give it the transient modifier
  + Does not create file automatically if it does not yet exist
  + Reflection happens when it decides what to serialize
  + Class and ALL of its parent classes whose fields we use must implement Serializable
* FIleIO
  + java.io
  + Java programs IO through streams
    - Stream is an abstraction that produces or consumes info
    - IO classes and methods can be applied to differrent devices
      * Input streams can abstract many different kinds of input”
        + Keyboard
        + File
        + Network socket
        + Etc
      * Same for output stream
        + Console
        + File
        + Network connection
  + We can either use
    - Byte Stream
      * Input and output for bytes
        + Reading and writing binary
      * Machine readable, byte code
      * Top of hierarchy-two abstract classes:
        + InputStream and OutputStream



* + - * For serializing (Serializable marker interface)
* Character Stream
* Input and output of characters (Unicode)
  + Human readable
  + FileReader and FileWriter

****

* **Reading and Writing Files**
  + Two most often-used stream classes
    - FileInputStream and FileOutputStream
      * create byte streams linked to files.
  + To open a file
    - Create an object of FileInputStream or FileOutputStream
    - specify the name of the file as an argument to the constructor
      * FileInputStream(String fileName) throws FileNotFoundException
      * FileOutputStream(String fileName) throws FileNotFoundException
  + When done with file, close it
    - .close()
  + To read from a file
    - Use read() from FileInputStream
      * Each time that it is called, it reads a single byte from the file and returns the byte as an integer value. read( ) returns –1 when the end of the file is encountered.
  + To write to a file
    - Use write() from FileOutoutStream
      * This method writes the byte specified by byteval to the file. Although byteval is declared as an integer, only the low-order eight bits are written to the file.