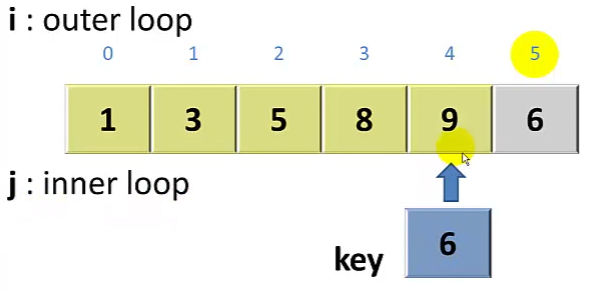
**Java 7 features**

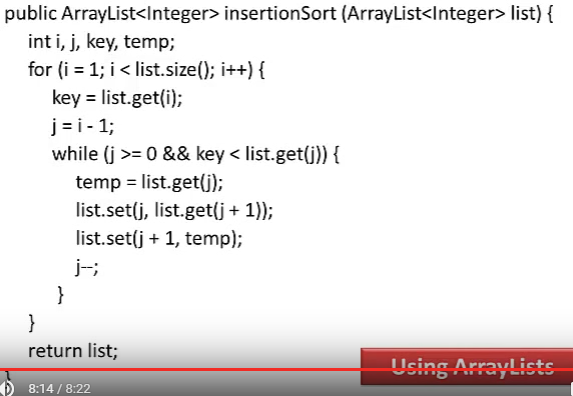
* String in switch statement (Java 7)
* Binary Literals (Java 7)
* The try-with-resources (Java 7)
* Caching Multiple Exceptions by single catch (Java 7)
* Underscores in Numeric Literals (Java 7)

|  |  |  |
| --- | --- | --- |
| **Feature** | **Prior to JDK7** | **In JDK 7** |
| **Type inference** | Map<String, List<String>> employeeRecords = new HashMap<String, List<String>>(); List<Integer> primes = new ArrayList<Integer>(); | Map<String, List<String>> employeeRecords = new HashMap<>(); List<Integer> primes = new ArrayList<>(); |
| **String in Switch** |  | String state = "NEW"; switch (day) {  case "NEW": System.out.println("Order is in NEW state"); break;  case "CANCELED": System.out.println("Order is Cancelled"); break;  case "REPLACE": System.out.println("Order is replaced successfully"); break;  case "FILLED": System.out.println("Order is filled"); break;  default: System.out.println("Invalid"); } |
| **Automatic Resource Management** | public static void main(String args[]) {  FileInputStream fin = null;  BufferedReader br = null;  try {  fin = new FileInputStream("info.xml");  br = new BufferedReader(new InputStreamReader(fin));  if (br.ready()) {  String line1 = br.readLine();  System.out.println(line1);  }  } catch (FileNotFoundException ex) {  System.out.println("Info.xml is not found");  } catch (IOException ex) {  System.out.println("Can't read the file");  } finally {  try {  if (fin != null) fin.close();  if (br != null) br.close();  } catch (IOException ie) {  System.out.println("Failed to close files");  }  }  } | public static void main(String args[]) {  try (FileInputStream fin = new FileInputStream("info.xml");  BufferedReader br = new BufferedReader(new InputStreamReader(fin));) {  if (br.ready()) {  String line1 = br.readLine();  System.out.println(line1);  }  } catch (FileNotFoundException ex) {  System.out.println("Info.xml is not found");  } catch (IOException ex) {  System.out.println("Can't read the file");  } } |
| **Underscore in numeric literals** | int billion = 1\_000\_000\_000; // 10^9 long creditCardNumber = 1234\_4567\_8901\_2345L; //16 digit number long ssn = 777\_99\_8888L; double pi = 3.1415\_9265; float pif = 3.14\_15\_92\_65f; | double pi = 3.\_1415\_9265; // underscore just after decimal point long creditcardNum = 1234\_4567\_8901\_2345\_L; //underscore at the end of number long ssn = \_777\_99\_8888L; //undersocre at the beginning |
| **Catching Multiple Exception Type in Single Catch Block** | try {  ...... } catch(ClassNotFoundException ex) {  ex.printStackTrace(); } catch(SQLException ex) {  ex.printStackTrace(); } | try {  ...... } catch(ClassNotFoundException|SQLException ex) {  ex.printStackTrace(); } |
| **More Precise Rethrowing of Exception** | public void obscure() throws Exception{  try {  new FileInputStream("abc.txt").read();  new SimpleDateFormat("ddMMyyyy").parse("12-03-2014");   } catch (Exception ex) {  System.out.println("Caught exception: " + ex.getMessage());  throw ex;  } } | public void precise() throws ParseException, IOException {  try {  new FileInputStream("abc.txt").read();  new SimpleDateFormat("ddMMyyyy").parse("12-03-2014");   } catch (Exception ex) {  System.out.println("Caught exception: " + ex.getMessage());  throw ex;  } } |

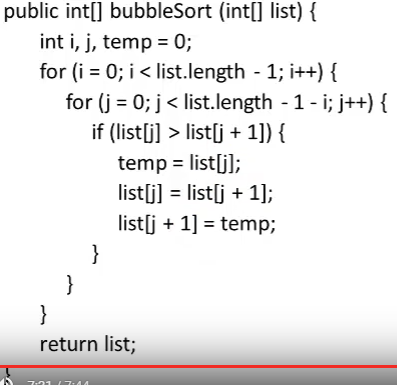
Sorting algorithms

* Insertion sort: O(n^2)
  + Check the left item of the original item.
  + If left item > original item, then swap.





* Bubble sort: O(n^2) where n is the number of items being sorted
  + Compare the first 2 items in the list.
  + If 1st > 2nd, then swap. Do this till end of the list.
  + Follow same steps in next iterations.



* Quick sort: O(n log(n))
  + Steps to implement
    - Choose an element, called pivot, from the list. Generally pivot can be the middle index element.
    - Reorder the list so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
    - Recursively apply the above steps to the sub-list of elements with smaller values and separately the sub-list of elements with greater values.

**Stack and Heap memory**

**Heap**:

* Java Heap space is used by java runtime to allocate memory to Objects and JRE classes.
* Whenever we create any **object, it’s always created in the Heap space**.

**Stack:**

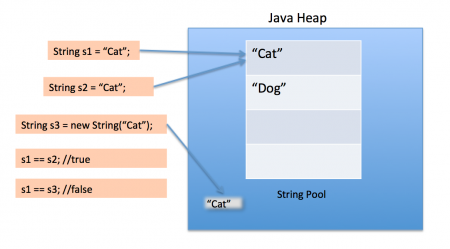
* Java Stack memory is used for execution of a thread.
* They contain method specific values that are short-lived and references to other objects in the heap that are getting referred from the method.
* As soon as method ends, the block becomes unused and become available for next method.

**Stack memory size is very less compared to Heap memory.**

**Stack vs Heap**

* Heap memory is used by all the parts of the application whereas stack memory is used only by one thread of execution.
* Whenever an object is created, it’s always stored in the Heap space and stack memory contains the reference to it. Stack memory only contains local primitive variables and reference variables to objects in heap space.
* Objects stored in the heap are globally accessible whereas stack memory can’t be accessed by other threads.
* Memory management in stack is done in LIFO manner whereas it’s more complex in Heap memory because it’s used globally. Heap memory is divided into Young-Generation, Old-Generation etc
* Stack memory is short-lived whereas heap memory lives from the start till the end of application execution.
* We can use **-Xms** and **-Xmx** JVM option to define the startup size and maximum size of heap memory. We can use **-Xss** to define the stack memory size.
* When stack memory is full, Java runtime throws java.lang.StackOverFlowError whereas if heap memory is full, it throws java.lang.OutOfMemoryError: Java Heap Space error.
* Stack memory size is very less when compared to Heap memory. Because of simplicity in memory allocation (LIFO), stack memory is very fast when compared to heap memory.

**String Pool** in java is a pool of Strings stored in [**Java Heap Memory**](https://www.journaldev.com/4098/java-heap-space-vs-stack-memory)**.**

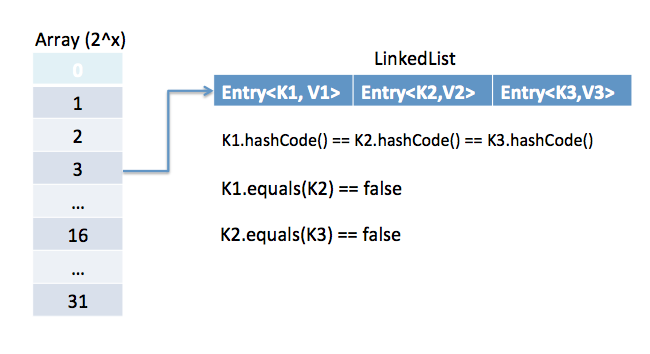


* String Pool is possible only because [String is immutable in Java](https://www.journaldev.com/802/string-immutable-final-java)
* When we use double quotes to create a String, it first looks for String with same value in the String pool, if found it just returns the reference else it creates a new String in the pool and then returns the reference.
* However using *new* operator, we force String class to create a new String object in heap space.
* We can use intern() method to put it into the pool or refer to other String object from string pool having same value.
* **String str = new String("Cat");**
  + In above statement, either 1 or 2 string will be created. If there is already a string literal “Cat” in the pool, then only one string “str” will be created in the pool.
  + If there is no string literal “Cat” in the pool, then it will be first created in the pool and then in the heap space, so total 2 string objects will be created.
* **String str = "abc"; or String str = new String ("abc");**
  + When we create a String using double quotes, it first looks for the String with same value in the JVM string pool, if found it returns the reference else it creates the String object and then place it in the String pool.
  + This way JVM saves a lot of space by using same String in different threads.
  + But if new operator is used, it explicitly creates a new String in the heap memory.
* **String vs StringBuffer vs StringBuilder**
  + String is immutable whereas StringBuffer and StringBuider are mutable classes.
  + StringBuffer is thread safe and synchronized whereas StringBuilder is not, thats why [StringBuilder is more faster than StringBuffer](https://www.journaldev.com/137/stringbuffer-vs-stringbuilder).
  + String concat + operator internally uses StringBuffer or StringBuilder class.
  + For String manipulations in non-multi threaded environment, we should use StringBuilder else use StringBuffer class.
* **Hashmap**
  + **extends AbstractMap class that implements Map interface, allows null key and null values, neither ordered nor threadsafe, it’s unsynchronized and allows null key and values, doesn't allow duplicate keys**
  + Stores **entries into multiple singly linked lists**, called buckets or bins. **Default number of bins is 16 and it’s always power of 2.**
  + Uses hashCode() and equals() methods on keys for get and put operations. **Immutable classes are better suitable for keys, for example String and Integer**.
  + HashMap has an inner class **Entry.** Entry class has key and value mapping stored as attributes.
  + **Instances of Entry class are stored in an array.**
  + **Single threaded:** 
    - **Hashmap**
  + **Multi threaded:** 
    - **ConcurrentHashMap class**
    - **Collections.synchronizedMap()**

|  |  |  |
| --- | --- | --- |
| **Constructor** | **initial capacity** | **load factor** |
| public HashMap() | 16 | 0.75 |
| public HashMap(int initialCapacity) | need to specify | 0.75 |
| public HashMap(int initialCapacity, float loadFactor) | need to specify | need to specify |
| public HashMap(Map<? extends K, ? extends V> m) | same as that of given map | same as that of given map |

* **Important methods**
  + **public V get(Object key):** Returns the value mapped to the specified key, or null if there is no mapping for the key.
  + **public V put(K key, V value):** Associates the specified value with the specified key in this map. If the map previously contained a mapping for the key, the old value is replaced.
  + **public V remove(Object key):** Removes the mapping for the specified key from this map if present.
* **Working**
  + HashMap works on hashing algorithm and uses hashCode() and equals() method on key for get and put operations.
  + use singly linked list to store elements
  + **Calling put method**:
    - Key object is checked for null. If key is null, value is stored in table[0] position. Because hash code for null is always 0.
    - Hash value is calculated using key’s hash code by calling its hashCode() method.
    - Hash value is used to calculate index in array for storing Entry object.
    - Now indexFor(hash, table.length) function is called to calculate exact index position for storing the Entry object.
    - Once **bucket is identified**, hashCode is used to check if there is already a key with same hashCode or not.
      * **Key available**: then equals() method is used on key.
        + **Equals return true**: then value is overwritten
        + Equals return false: a new mapping is made to this singly linked list bucket.
      * Key not available: then mapping is inserted into the bucket.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **PUT operation** | | | | | | |
| Using hashcode() method, is bucket available? | Yes | check if there is already a key with same hashCode? | Yes | equals() method is used on key. Returns? | TRUE | value is overwritten |
| FALSE | new mapping is made to this singly linked list bucket |
| No | mapping is inserted into the bucket. |  |  |
| No |  |  |  |  |  |



* **Important points**
  + Data structure to store Entry objects is an array named table of type Entry.
  + A particular index location in array is referred as bucket, because it can hold the first element of a LinkedList of Entry objects.
  + Key object’s hashCode() is required to calculate the index location of Entry object.
  + Key object’s equals() method is used to maintain uniqueness of Keys in map.
  + Value object’s hashCode() and equals() method are not used in HashMap’s get() and put() methods.
  + Hash code for null keys is always zero, and such Entry object is always stored in zero index in Entry[].
* **Ways to iterate**
  + by using keySet and iterator
  + by using entrySet and iterator
  + by using entrySet and enhanced for loop
  + by using keySet and get() method

**ConcurrentHashMap**is the class that is similar to HashMap but **works fine** when you try to **modify your map at runtime**.

**Concurrent collection classes: CopyOnWriteArrayList, ConcurrentHashMap, CopyOnWriteArraySet**

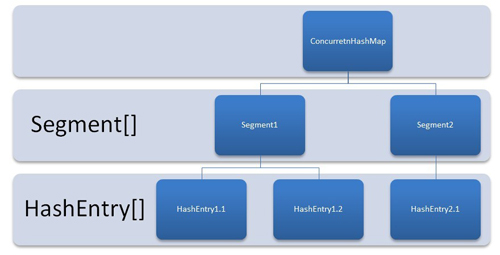
Chosing the right type of collection based on the need, for example **if size is fixed**, we might want to use **Array over ArrayList**. If we have to ***iterate over the Map in order of insertion***, we need to use ***TreeMap***. If we **don’t want duplicates**, we should use **Set**.

There are two ways we could synchronized [HashMap](http://crunchify.com/java-hashmap-containskeyobject-key-and-containsvalueobject-value-check-if-key-exists-in-map/" \t "_blank)

* Java Collections synchronizedMap() method
* Use ConcurrentHashMap

**ConcurrentHashMap**

* You should use ConcurrentHashMap when you need very high concurrency in your project.
* It is thread safe without synchronizing the whole map.
* Reads can happen very fast while write is done with a lock.
* There is no locking at the object level.
* The locking is at a much finer granularity at a hashmap bucket level.
* ConcurrentHashMap doesn’t throw a ConcurrentModificationException if one thread tries to modify it while another is iterating over it.
* ConcurrentHashMap uses multitude of locks.
* **Internal structure**
  + ConcurrentHashMap has an addition concept of segments.
  + It will be easier to understand it you think of one segment equal to one HashMap [conceptually].
  + A concurrentHashMap is divided into number of segments [default 16] on initialization.
  + ConcurrentHashMap allows similar number (16) of threads to access these segments concurrently so that each thread work on a specific segment during high concurrency.
  + This way, if when your key-value pair is stored in segment 10; code does not need to block other 15 segments additionally.
  + This structure provides a very high level of concurrency.
  + In other words, **ConcurrentHashMap uses a multitude of locks, each lock controls one segment of the map**. When setting data in a particular segment, the lock for that segment is obtained. So essentially **update operations are synchronized**.



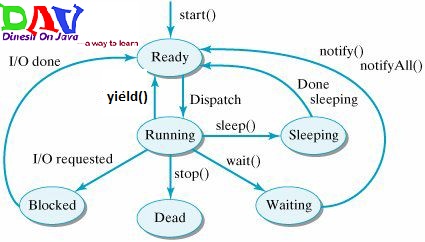
**SynchronizedHashMap**

* Synchronization at Object level.
* Every read/write operation needs to acquire lock.
* Locking the entire collection is a performance overhead.
* This essentially gives access to only one thread to the entire map & blocks all the other threads.
* It may cause contention.
* SynchronizedHashMap returns Iterator, which fails-fast on concurrent modification.

**Java Cursors**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Enumeration** | **Iterator** | **ListIterator** | **Spliterator** |
| **main methods** | **hasMoreElements()** and **nextElement()** | **boolean hasNext():**Returns true if the iteration has more elements. **E next():** Returns the next element in the iteration. **default void remove():** Removes from the underlying collection the last element returned by this iterator. default void **forEachRemaining(Consumer action):** Performs the given action for each remaining element until all elements have been processed or the action throws an exception. | Forward Direction Iteration methods  **hasNext()) next() nextIndex()**   Backward Direction Iteration methods  **hasPrevious() previous() previousIndex()** | **tryAdvance()** method to iterate elements individually in multiple Threads to support Parallel Processing. **forEachRemaining()** method to iterate elements sequentially in a single Thread. **trySplit()** method to divide itself into Sub-Spliterators to support Parallel Processing. |
| **supported processing** | Sequential | Sequential | Sequential | Sequential and Parallel |
| **supported API** | Collection | Collection | Collection | Collection and Stream |
| **operations supported** | **only READ** | both READ and DELETE | supports all four operations: CRUD (CREATE, READ, UPDATE and DELETE) | supports all four operations: CRUD (CREATE, READ, UPDATE and DELETE) |
| **direction iteration** | supports only Forward Direction iteration | supports only Forward Direction iteration | Supports both Forward Direction and Backward Direction iterations. | Supports both Forward Direction and Backward Direction iterations. |
| **Details** |  | Each and every Collection class has the following iterator() method to iterate it’s elements. Iterator<E> iterator() It returns an iterator over the elements available in the given Collection object. | iterate elements one-by-one from a List implemented object only. Ex: ArrayList, CopyOnWriteArrayList, LinkedList, Stack, Vector etc | Unlike Iterator and ListIterator, it supports Parallel Programming functionality. Unlike Iterator and ListIterator, it supports both Sequential and Parallel Processing of data. Compare to other Iterators, it provides better performance. |
| **Disadvantages** | since Java 1.0, legacy interface, only for Collection Legacy classes, very lengthy method names, supports only READ operation, supports only Forward Direction iteration, not recommended to use it in new code base or projects. | does NOT support CREATE and UPDATE, supports only Forward direction iteration, supports only Sequential iteration,  Compare to Spliterator, it does NOT support better performance to iterate large volume of data. | Only List implementation classes, not applicable for whole Collection API,  does NOT support Parallel iteration of elements, Compare to Spliterator, it does NOT support better performance to iterate large volume of data. | **We can NOT use this Iterator for Map implemented classes.** |

**Thread lifecycle**



**Java thread dump is very helpful in analyzing bottlenecks in the application and**[**deadlock**](https://www.journaldev.com/1058/deadlock-in-java-example)**situations.**

**Tools: VisualVM Profiler, jstack**

**Thread dump** is the list of all the threads, every entry shows information about thread which includes following in the order of appearance.

* + **Thread Name**: Name of the Thread
  + **Thread Priority**: Priority of the thread
  + **Thread ID**: Represents the unique ID of the Thread
  + **Thread Status**: Provides the current [thread state](https://www.journaldev.com/1044/thread-life-cycle-in-java-thread-states-in-java), for example RUNNABLE, WAITING, BLOCKED. While analyzing deadlock look for the blocked threads and resources on which they are trying to acquire lock.
  + **Thread callstack**: Provides the vital stack information for the thread. This is the place where we can see the locks obtained by Thread and if it’s waiting for any lock.

**Ways to implement singleton**

|  |  |  |
| --- | --- | --- |
| **Way** | **Code** | **Drawback** |
| **Classic java Singleton (static reference to the singleton instance and a Private constructor)** | public class Singleton {  private static Singleton instance;  private Singleton (){}    public static Singleton getInstance() {  if (instance == null) {  instance = new Singleton();  }  return instance;  } } | Not threadsafe |
| **thread-safe singleton pattern in java using Synchronization** | public synchronized static Singleton getInstance() {  if(singleton == null) {  singleton = new Singleton();  }  return singleton; } | We actually only need the method be synchronized the first time when it is called . Because synchronization is very expensive, synchronized methods can run up to 100 times slower than unsynchronized methods, we need to introduce a performance enhancement that only synchronize the singleton assignment in getInstance(). |
| **Synchronize the critical code only** | public static Singleton getInstance() {  if(singleton == null) {  synchronized(Singleton.class) {   singleton = new Singleton();  }  }  return singleton; } | However, this method is not thread-safe. Consider the following scenario: Thread 1 enters the synchronized block, and, before it can assign the singleton member variable, the thread is preempted. Then another thread can enter the if block. The second thread will wait for the first thread to finish, but we will still wind up with two distinct singleton instances. Is there a way to fix this problem? |
| **Double-checked locking** | public static Singleton getInstance() {  if(singleton == null) {  synchronized(Singleton.class) {  if(singleton == null) {  singleton = new Singleton();  }  }  }  return singleton; } | Unfortunately, double-checked locking is not guaranteed to work because the compiler is free to assign a value to the singleton member variable before the singleton’s constructor is called. If that happens, Thread 1 can be preempted after the singleton reference has been assigned, but before the singleton is initialized, so Thread 2 can return a reference to an uninitialized singleton instance. |
| **Double-checked locking with volatile keyword** | public class Singleton {  private volatile static Singleton instance; // volatile  private Singleton (){}    public static Singleton getSingleton() {  if (instance == null) {   synchronized (Singleton.class) {  if (instance == null) {   instance = new Singleton();  }  }  }  return instance;  } } | The volatile prevents memory writes from being re-ordered, making it impossible for other threads to read uninitialized fields of your singleton through the singleton’s pointer.  However, we should avoid use the volatile based method, as it is hard to understand and it is easy to make mistakes. |
| **Thread-safe but not lay initialized** | public class Singleton{  //the variable will be created when the class is loaded   private static final Singleton instance = new Singleton();  private Singleton(){}  public static Singleton getInstance(){  return instance;  } } | This method is thread-safe, but it is not lazy initialized . The singleton object is created as soon as the class is loaded. |
| **The ultimate Thread-safe and efficient singleton pattern in Java** | // Correct lazy initialization in Java  @ThreadSafe class Singleton {  private static class SingletonHolder {  public static Singleton instance = new Singleton();  }    public static Singleton getInstance() {  return SingletonHolder.instance;  } } | Advantage: The method is recommend by effective java. It is lazy initialized, and multiple-thread safe. |

