Java virtual machine (JVM)

Implementation of the Java Virtual Machine Specification, interprets compiled Java binary code (called bytecode) for a computer's processor (or "hardware platform") so that it can perform a Java program's instructions. Java was designed to allow application programs to be built that could be run on any platform without having to be rewritten or recompiled by the programmer for each separate platform. A Java virtual machine makes this possible because it is aware of the specific instruction lengths and other particularities of the platform.

The Java Virtual Machine Specification defines an abstract -- rather than a real -- machine or processor. The Specification specifies an instruction set, a set of registers, a stack, a "garbage heap," and a method area. Once a Java virtual machine has been implemented for a given platform, any Java program (which, after compilation, is called bytecode) can run on that platform. A Java virtual machine can either interpret the bytecode one instruction at a time (mapping it to a real processor instruction) or the bytecode can be compiled further for the real processor using what is called a just-in-time compiler.

Java Runtime Environment (**JRE**) is a software package that contains what is required to run a Java program. It includes a Java Virtual Machine implementation together with an implementation of the Java Class Library.

Java Development Kit (JDK) is a superset of a JRE and contains tools for Java programmers, e.g. a javac compiler.

Locality

e.g., Arrays have better cache locality than LLs

```
+----------+
| CPU      |  <<-- Our beloved CPU, superfast and always hungry for more data.
+----------+
|L1 - Cache|  <<-- works at 100% of CPU speed (fast)
+----------+
|L2 - Cache|  <<-- works at 25% of CPU speed (medium)
+----+-----+
   |
   |     <<-- This thin wire is the memory bus, it has limited bandwidth.
+----+-----+  <<-- works at 10% of CPU speed.
| main-mem |  <<-- The main memory is big but slow (because we are cheap-skates)
+----------+
   |
   |    <<-- Even slower wire to the harddisk
+----+-----+
```

```
| harddisk | <<-- Works at 0,001% of CPU speed
+----------+
```

Spatial Locality

In this diagram, the closer data is to the CPU the faster the CPU can get at it.
This is related to Spacial Locality. Data has spacial locality if it is located close together in memory.
Because of the cheap-skates that we are RAM is not really Random Access, it is really Slow if random, less slow if accessed sequentially Access Memory SIRLSIAS-AM.
That is why it is smart to keep related data close together, so you can do a sequential read of a bunch of data and save time.

Temporal locality

Data stays in main-memory, but it **cannot** stay in the cache, or the cache would stop being useful. One the most recently used data can be found in the cache; old data gets pushed out.
This is related to temporal locality. Data has strong temporal locality if it is accessed at the same time.
This is important because if item A is in the cache (good) than Item B (with strong temporal locality to A) is very likely to also be in the cache.

Important Algorithms (http://algs4.cs.princeton.edu/lectures/)

Sorting -> Mege, Quick
Digraph -> DFS, BFS, Topological
UnDigraph -> DFS, BFS, Connected components
Min Spanning Trees -> Greedy, Kruskal, Prim
Shortest Paths -> Dijkstra, Bellman-Ford
Max-Flow -> Ford-Fulkerson

Sorting Algo

Quicksort vs Heapsort – Quicksort is O(nlogn) on average BUT worst case is n^2 if pivot is bad. Heapsort has worst case of O(nlogn) on average.

Quicksort vs Mergesort – both O(nlogn) avg case but quicksort is in-place so doesn't use space.

Big O and Avg time vs Amortalized

Big O notation is used to classify algorithms according to how their running time or space requirements grow as the input size grows. Only uses upper bound (worst case). Measures an algorithm's efficiency.

- **Worst case Running Time**: The behavior of the algorithm with respect to the worst possible case of the input instance. The worst-case running time of an algorithm is an upper bound on the running time for any input. Knowing it

gives us a guarantee that the algorithm will never take any longer. There is no need to make an educated guess about the running time.

- **Average case Running Time**: The expected behavior when the input is randomly drawn from a given distribution. The average-case running time of an algorithm is an estimate of the running time for an "average" input. Computation of average-case running time entails knowing all possible input sequences, the probability distribution of occurrence of these sequences, and the running times for the individual sequences. Often it is assumed that all inputs of a given size are equally likely.
- **Amortized Running Time** Here the time required to perform a sequence of (related) operations is averaged over all the operations performed. Amortized analysis can be used to show that the average cost of an operation is small, if one averages over a sequence of operations, even though a simple operation might be expensive. Amortized analysis guarantees the average performance of each operation in the worst case.

Primitive vs Reference Type

```
int a = 77;
Person person = new Person();
```

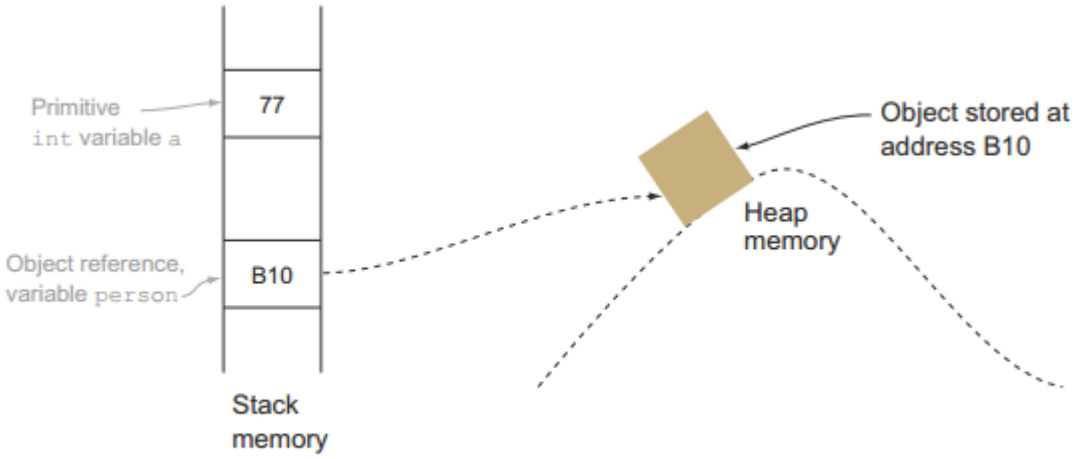


Figure 2.13 Primitive variables store the actual values, whereas object reference variables store the addresses of the objects they refer to.

In-Place and Stable

An *in-place* sorting algorithm directly modifies the list that it receives as input instead of creating a new list that is then modified. An example of an in-place sorting algorithm is bubble sort that simply swaps the elements of the array received as input.

A *stable* sorting algorithm leaves elements in the list that have equal sorting keys at the same places that they were in the input. Bubble sort is a stable algorithm, whereas e.g. quick sort isn't.

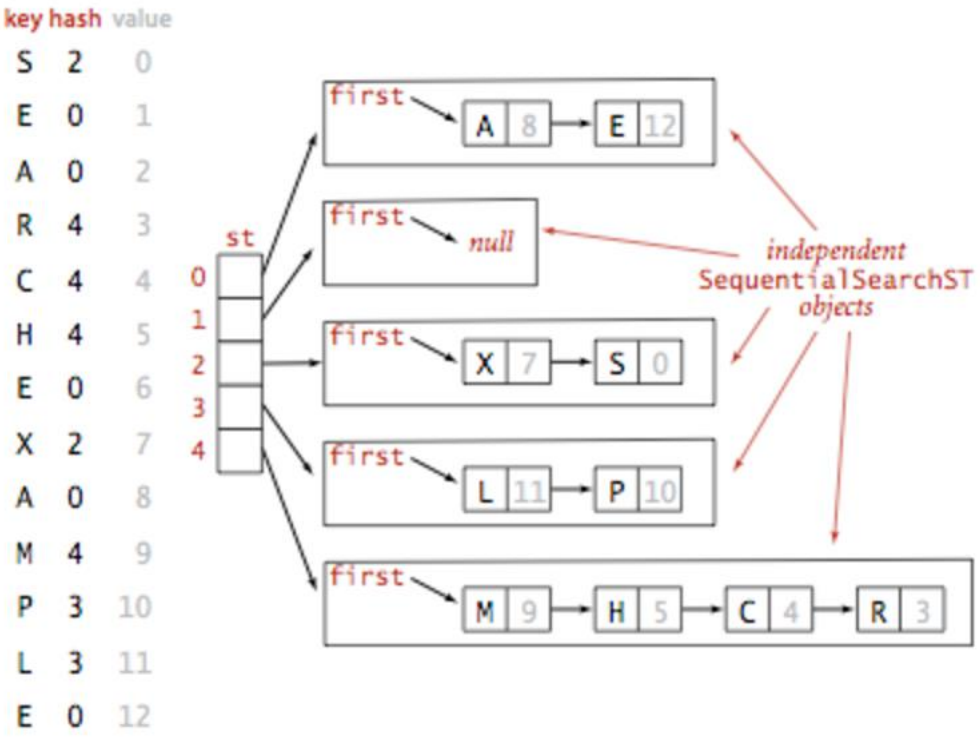
Hash Functions and Collision

A hash function takes a group of characters (called a key) and maps it to a value of a certain length (called a hash value or hash). The hash value is representative of the original string of characters, but is normally smaller than the original.

Hashing is done for indexing and locating items in databases because it is easier to find the shorter hash value than the longer string. Hashing is also used in encryption.

To handle collision ->

1) separate chaining - A straightforward approach to collision resolution is to build, for each of the M array indices, a linked list of the key-value pairs whose keys hash to that index. The basic idea is to choose M to be sufficiently large that the lists are sufficiently short to enable efficient search through a two-step process: hash to find the list that could contain the key, then sequentially search through that list for the key.



Hashing with separate chaining for standard indexing client

2) linear probing- Another approach to implementing hashing is to store N key-value pairs in a hash table of size M > N, relying on empty entries in the table to help with with

collision resolution. Such methods are called *open-addressing* hashing methods. The simplest open-addressing method is called *linear probing*: when there is a collision (when we hash to a table index that is already occupied with a key different from the search key), then we just check the next entry in the table (by incrementing the index). There are three possible outcomes:

- key equal to search key: search hit
- empty position (null key at indexed position): search miss
- key not equal to search key: try next entry

Runtime complexity is mostly O(1) on average. However, it suffers from O(n) worst case due to collision and from rehashing once hash table passed its load balance.

Hashtable vs HashMap vs LinkedHashMap vs Hashset

Hashtable

Hashtable is basically a data structure to retain values of key-value pair.

- It *didn't allow null* for both key and value. You will get NullPointerException if you add null value.
- It is synchronized so it comes with its cost. Only one thread can access in one time

```
1  Hashtable<Integer,String>; cityTable = new Hashtable<Integer,String>();
2  cityTable.put(1, "Lahore");
3  cityTable.put(2, "Karachi");
4  cityTable.put(3, null); /* NullPointerEcxeption at runtime*/
5
6  System.out.println(cityTable.get(1));
7  System.out.println(cityTable.get(2));
8  System.out.println(cityTable.get(3));
```

HashMap

Like Hashtable it also accepts key value pair.

- It *allows null* for both key and value
- It is unsynchronized so come up with better performance
- No ordering guaranteed

```
1  HashMap<Integer,String> productMap = new HashMap<Integer,String>();
2  productMap.put(1, "Keys");
3  productMap.put(2, null);
```

Linked HashMap

Key are maintained in insertion order.

HashSet

HashSet *does not allow duplicate values*. It provides add method rather put method. You also

use its contain method to check whether the object is already available in HashSet. HashSet can be used where you want to maintain a unique list.

```
1    HashSet<String> stateSet = new HashSet<String>();
2    stateSet.add ("CA");
3    stateSet.add ("WI");
4    stateSet.add ("NY");
5
6    if (stateSet.contains("PB")) /* if CA, it will not add but shows following message*/
7    System.out.println("Already found");
8    else
9    stateSet.add("PB");
```

LinkedHashMap
Will iterate in the order in which entries were put into map.

To Iterate:

Map.Entry is a key and its value combined into one class. The allows you to iterate over Map.entrySet() instead of having to iterate over Map.keySet(), then getting the value for each key. A better way to write what you have is:

```
for (Map.Entry<String, JButton> entry : listbouton.entrySet())
{
 String key = entry.getKey();
 JButton value = entry.getValue();

 this.add(value);
}
```

If a method returns a collection of objects that implement an interface, you don't need to worry about what the underlying object is. In this particular case the object implementing the interface Map.Entry is an inner class in whatever Map implementation you're using (in this case it's built into HashMap). All you need to know is that the object meets the contract specified by the interface, and you can therefore call the methods specified by the interface.

List vs. Set

List (such as ArrayList) can contain duplicates.
Set (such as HashSet) cannot contain duplicates.

Linked List vs. Array

Both Arrays and Linked List can be used to store linear data of similar types

Pros of Linked Lists:
(1) The size of the arrays is fixed: So we must know the upper limit on the number of elements in

advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage, and in practical uses, upper limit is rarely reached.
(2) Inserting a new element in an array of elements is expensive, because room has to be created for the new elements and to create room existing elements have to shifted.
For example, suppose we maintain a sorted list of IDs in an array id[].
id[] = [1000, 1010, 1050, 2000, 2040, …..].
And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000).
Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved.

So Linked list provides following two advantages over arrays
1) Dynamic size
2) Ease of insertion/deletion

Linked lists have following drawbacks:
1) Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists.
2) Extra memory space for a pointer is required with each element of the list.
3) Arrays have better cache locality that can make a pretty big difference in performance.

DFS vs BFS

That heavily depends on the structure of the search tree and the number and location of solutions (aka searched-for items). If you know a solution is not far from the root of the tree, a breadth first search (BFS) might be better. If the tree is very deep and solutions are rare, depth first search (DFS) might take an extremely long time, but BFS could be faster. If the tree is very wide, a BFS might need too much memory, so it might be completely impractical. If solutions are frequent but located deep in the tree, BFS could be impractical. If the search tree is very deep you will need to restrict the search depth for depth first search (DFS), anyway (for example with iterative deepening).

Iterable vs Iterator

An `Iterable` is a simple representation of a series of elements that can be iterated over. It does not have any iteration state such as a "current element". Instead, it has one method that produces an `Iterator`.
An `Iterator` is the object with iteration state. It lets you check if it has more elements using `hasNext()` and move to the next element (if any) using `next()`.
Typically, an `Iterable` should be able to produce any number of valid `Iterator`s.

Handling Exceptions

1) Catching exceptions using try-catch or multi try-catch

2) Using throws/throw keywords
3)Finally block

Resources:
https://www.tutorialspoint.com/java/java_exceptions.htm
http://beginnersbook.com/2013/04/difference-between-throw-and-throws-in-java/
http://beginnersbook.com/2013/04/java-throws/