Core Java Foundation : Day04

Akshita Chanchlani

Collection Framework

- Every value/data stored in data structure is called element.
- Framework is library of reusable classes/interfaces that is used to develop application.
- **Library of reusable data structure classes that is used to develop java application is called collection framework.**
- Main purpose of collection framework is to manage data in RAM efficiently.
- Consider following Example:
    1. Person has-a birthdate
    2. Employee is a person
- In java, collection instance do not contain instances rather it contains reference of instances.
- If we want to use collection framework them we should **import java.util** package.

Collection Framework

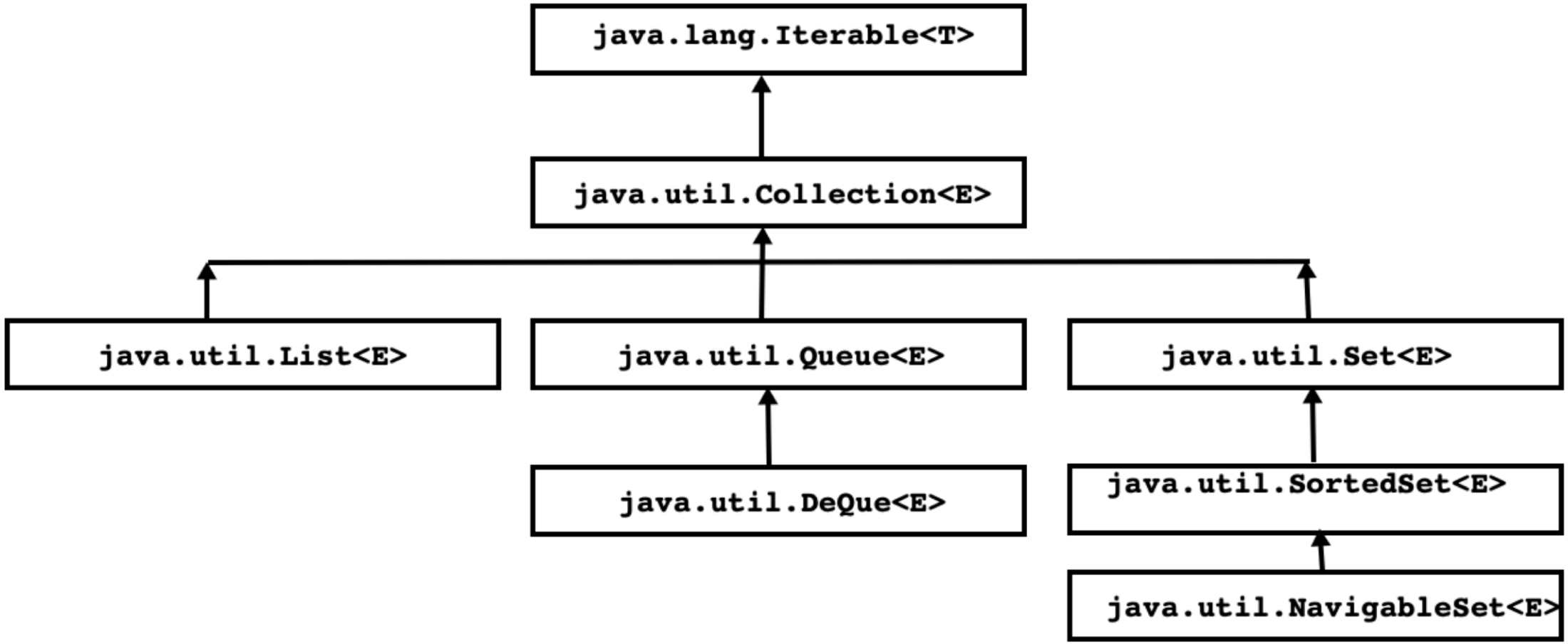
Java collection framework includes the following:

- **Interfaces**
  - Interface in Java refers to the abstract data types. They allow Java collections to be manipulated independently from the details of their representation. Also, they form a hierarchy in object-oriented programming languages.

- **Classes**
  - Classes in Java are the implementation of the collection interface. It basically refers to the data structures that are used again and again.

- **Algorithm**
  - Algorithm refers to the methods which are used to perform operations such as searching and sorting, on objects that implement collection interfaces

Collection Interface Hierarchy



Collection Framework Interface Hierarchy

Collection Interface

Collection Interface	This enables you to work with groups of objects; it is at the top of the collections hierarchy.
List	It extends Collection and an instance of List stores an ordered collection of elements.
Set	This extends Collection to handle sets, which must contain unique elements.
SortedSet	This extends Set to handle sorted sets.
Map	This maps unique keys to values.

Collection Interface Example

```
// ArrayList
      List a1 = new ArrayList();
      a1.add("sunbeam");
      a1.add("infotech");
      a1.add("pune");
      System.out.println(" ArrayList Elements");
      System.out.print("\t" + a1);

      // LinkedList
      List l1 = new LinkedList();
      l1.add(10);
      l1.add(20);
      l1.add(30);
      System.out.println();
      System.out.println(" LinkedList Elements");
      System.out.print("\t" + l1);
```

```
// HashSet
      Set s1 = new HashSet();
      s1.add("e1");
      s1.add("e2");
      s1.add("e3");
      System.out.println();
      System.out.println(" Set Elements");
      System.out.print("\t" + s1);

      // HashMap
      Map m1 = new HashMap();
      m1.put("key1", "10");
      m1.put("key2", "20");
      m1.put("key3", "30");
      m1.put("key4", "40");
      System.out.println();
      System.out.println(" Map Elements");
      System.out.print("\t" + m1);
```

List : ArrayList and LinkedList

```
public class main_prg {

    public static void main(String[] args) {
        List a1 = new ArrayList();
        a1.add("Akshita");
        a1.add("Chanchlani");

        System.out.println(" ArrayList Elements");
        System.out.print("\t" + a1);

        List l1 = new LinkedList();
        l1.add("Sunbeam");
        l1.add("Infotech");

        System.out.println();
        System.out.println(" LinkedList Elements");
        System.out.print("\t" + l1);
    }
}
```

Set : HashSet, TreeSet, LinkedHashSet

```
public class main_prg {

    public static void main(String[] args) {
        List a1 = new ArrayList();
        a1.add("Akshita");
        a1.add("Chanchlani");

        System.out.println(" ArrayList Elements");
        System.out.print("\t" + a1);

        List l1 = new LinkedList();
        l1.add("Sunbeam");
        l1.add("Infotech");

        System.out.println();
        System.out.println(" LinkedList Elements");
        System.out.print("\t" + l1);
    }
}
```

HashSet

If you don't want to maintain insertion order but want to store unique objects.

LinkedHashSet

If you want to maintain the insertion order of elements then you can use LinkedHashSet.

TreeSet

If you want to sort the elements according to some Comparator then use TreeSet.

SortedSet

The SortedSet interface extends Set and declares the behavior of a set sorted in an ascending order.

```
// Create the sorted set
      SortedSet set = new TreeSet();

      // Add elements to the set
      set.add(“r");
      set.add(“q");
      set.add(“p");

      // Iterating over the elements in the set
      Iterator it = set.iterator();

      while (it.hasNext()) {
         // Get element
         Object element = it.next();
         System.out.println(element.toString());
```

Map

The Map interface maps unique keys to values. A key is an object that you use to retrieve a value at a later date.

```
Map m1 = new HashMap();
    m1.put("key1", "10");
    m1.put("key2", "20");
    m1.put("key3", "30");
    m1.put("key4", "40");

    System.out.println();
    System.out.println(" Map Elements");
    System.out.print("\t" + m1);
```

Collection<E>

- Collection<E> is interface declared in java.util package.
- It is sub interface of Iterable interface.
- It is **root interface** in collection framework interface hierarchy.
- Default methods of Collection interface
  1. default Stream<E> **stream**()
  2. default Stream<E> **parallelStream**()
  3. default boolean **removeIf**(Predicate<? super E> filter)

Collection<E>

- Abstract Methods of Collection Interface

1. boolean **add**(E e)
2. boolean **addAll**(Collection<? extends E> c)
3. void **clear**()
4. boolean **contains**(Object o)
5. boolean **containsAll**(Collection<?> c)
6. boolean **isEmpty**()
7. boolean **remove**(Object o)
8. boolean **removeAll**(Collection<?> c)
9. boolean **retainAll**(Collection<?> c)
10. int **size**()
11. Object[] **toArray**()
12. <T> T[] **toArray**(T[] a)

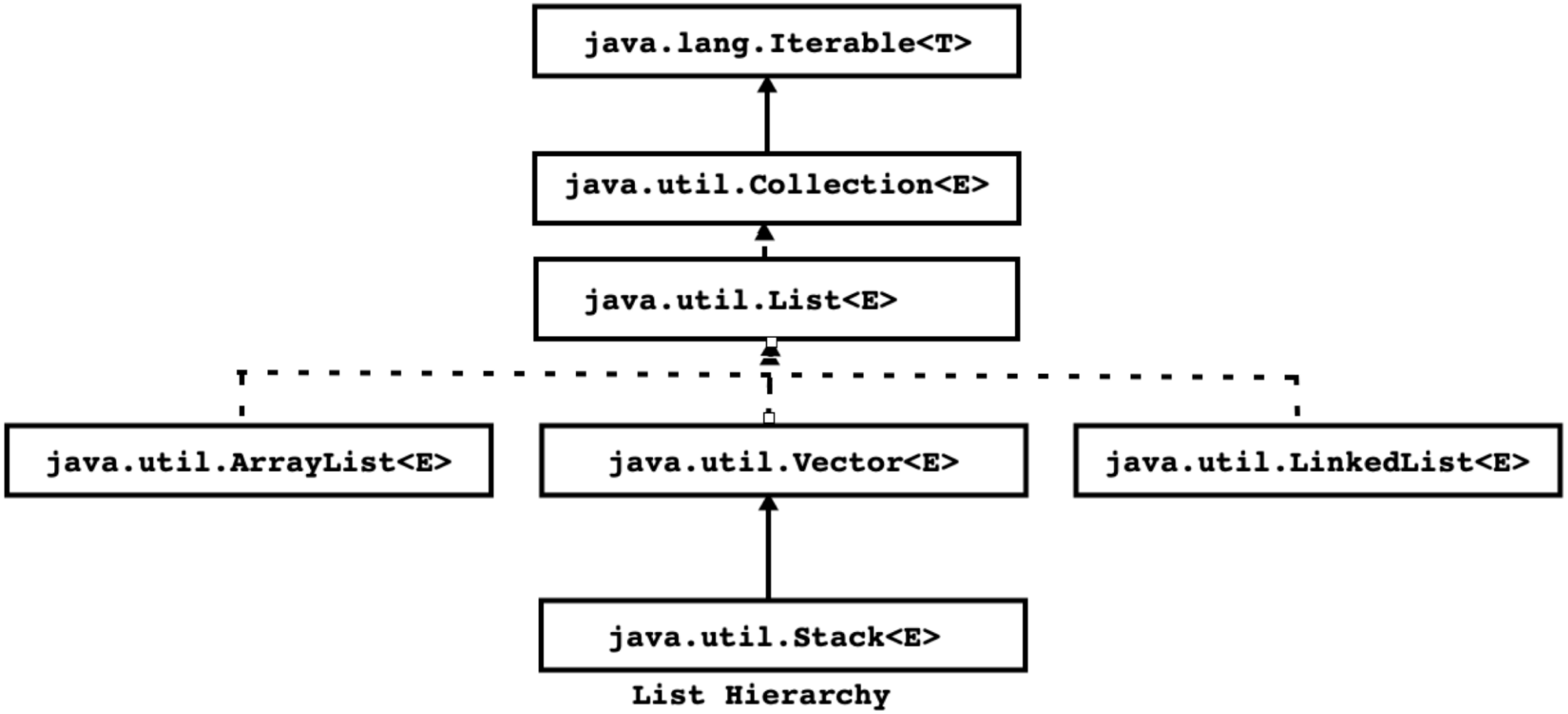
List<E>

- It is sub interface of java.util.Collection interface.
- It is ordered/sequential collection.
- ArrayList, Vector, Stack, LinkedList etc. implements List interface. It generally referred as "List collections".
- List collection can contain duplicate element as well multiple null elements.
- Using integer index, we can access elements from List collection.
- We can traverse elements of List collection using Iterator as well as ListIterator.
- It is introduced in jdk 1.2.
- **Note:** If we want to manage elements of non final type **inside** List collection then **non final type** should **override "equals" method.**

List<E>

- Abstract methods of List Interface
  1. void **add**(int index, E element)
  2. boolean **addAll**(int index, Collection<? extends E> c)
  3. E **get**(int index)
  4. int **indexOf**(Object o)
  5. int **lastIndexOf**(Object o)
  6. ListIterator<E> **listIterator**()
  7. ListIterator<E> **listIterator**(int index)
  8. E **remove**(int index)
  9. E **set**(int index, E element)
  10. List<E> **subList**(int fromIndex, int toIndex)

List Interface Hierarchy



List Hierarchy

ArrayList<E>

- It is resizable array.
- It implements List<E>, RandomAccess, Cloneable, Serializable interfaces.
- It is List collection.
- It is unsynchronized collection. Using"Collections.synchronizedList" method, we can make it synchronized.
  - List list = Collections.synchronizedList(new ArrayList(...));
- Initial capacity of ArrayList is 10. If ArrayList is full then its capacity gets increased by half of its existing capacity.

The elements stored in the ArrayList class can be randomly accessed.

Vector<E>

- It is resizable array.
- It implements List<E>, RandomAccess, Cloneable, Serializable.
- It is List collection.
- It is synchronized collection which means only one thread at a time can access the code.
- Default capacity of vector is 10. If vector is full then its capacity gets increased by its existing capacity.
- We can traverse elements of vector using Iterator, ListIterator as well as Enumeration.
- It is introduced in jdk 1.0.
- **Note:** If we want to manage elements of non final type inside Vector then non final type should override "equals" method.

Synchronized Collections

- 1. **Vector**
- 2. **Stack**(Sub class of Vector)
- 3. **Hashtable**
- 4. **Properties**(Sub class of Hashtable)

Enumeration<E>

- It is interface declared in java.util package.
- Methods of Enumeration I/F
    1. boolean hasMoreElements()
    2. E nextElement()
- It is used to traverse collection only in forward direction. During traversing, we can add, set or remove element from collection.
- It is introduced in jdk 1.0.
- "public Enumeration<E> elements()" is a method of Vector class.

```
Integer element = null;
Enumeration<Integer> e = v.elements();
while( e.hasMoreElements()){
    element = e.nextElement();
    System.out.println(element);
}
```

Iterator<E>

- It is a interface declared in java.util package.
- It is used to traverse collection only in forward direction. During traversing, we can not add or set element but we can remove element from collection.
- Methods of Iterator
  1. boolean hasNext()
  2. E next()
  3. default void remove()
  4. default void forEachRemaining(Consumer<? super E> action)
- It is introduced in jdk 1.2

```
Integer element = null;
Iterator<Integer> itr = v.iterator();
while( itr.hasNext()){
    element = itr.next();
    System.out.  println(element);
}
```

ListIterator<E>

- It is sub interface of Iterator interface.
- It is used to traverse only List Collection in bidirectional.
- During traversing, we can add, set as well as remove element from collection.
- It is introduced in jdk 1.2
- Methods of ListIterator
    1. boolean hasNext()
    2. E next()
    3. boolean hasPrevious()
    4. E previous()
    5. void add(E e)
    6. void set(E e)
    7. void remove()

ListIterator<E>

```
Integer element = null;
ListIterator<Integer> itr = v.listIterator();
while( itr.hasNext()){
    element = itr.next();
    System.out.print(element+"   ");
}

while( itr.hasPrevious()){
    element = itr.previous();
    System.out.print(element+"   ");
}
```

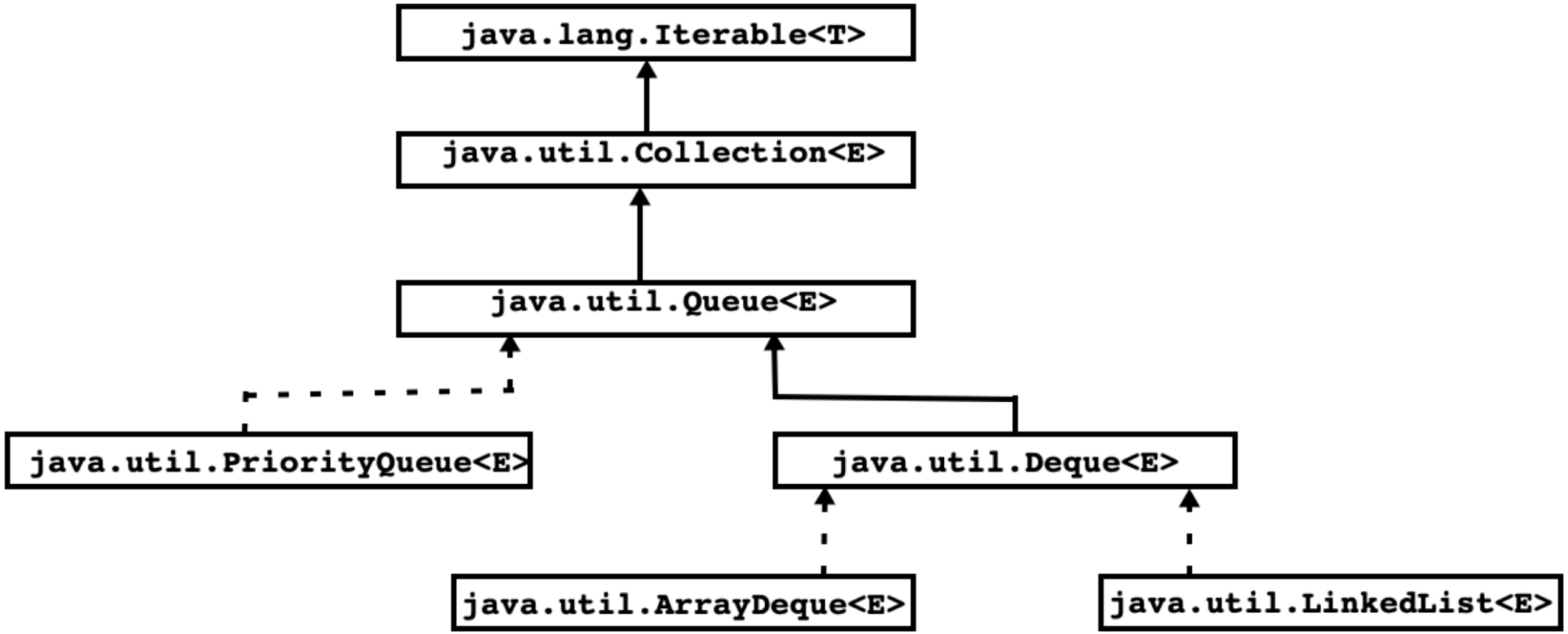
Stack<E>

- It is linear data structure which is used to manage elements in Last In First Out order.
- It is sub class of Vector class.
- It is synchronized collection.
- It is List Collection.
- Methods of Stack class
    1. public boolean empty()
    2. public E push(E item)
    3. public E peek()
    4. public E pop()
    5. public int search(Object o)
- * Since it is synchronized collection, it slower in performance.
- * For high performance we should use ArrayDeque class.

LinkedList<E>

- It is a List collection.
- It implements List<E>, Deque<E>, Cloneable and Serializable interface.
- Its implementation is depends on Doubly linked list.
- It is unsynchronized collection. Using Collections.synchronizedList() method, we can make it synchronized.
  - **List list = Collections.synchronizedList(new LinkedList(...));**
- It is introduced in jdk 1.2.
- Note : If we want to manage elements of non-final type inside LinkedList then non final type should override "equals" method.
- Instantiation
  - **List<Integer> list = new LinkedList<>();**

Queue Interface Hierarchy



Queue Hierarchy

Queue<E>

- It is interface declared in java.util package.
- It is sub interface of Collection interface.
- It is introduced in jdk 1.5

Summary of Queue methods

	Throws exception	*Returns special value*
Insert	`add(e)`	`offer(e)`
Remove	`remove()`	`poll()`
Examine	`element()`	`peek()`

Queue<E>

```
Queue<Integer> que = new ArrayDeque<>();
 que.offer(10);
 que.offer(20);
 que.offer(30);

 Integer ele = null;
 while( !que.isEmpty()){
     ele = que.peek();
     //TODO : processing
     que.poll();
 }
```

```
 Queue<Integer> que = new ArrayDeque<>();
 que.add(10);
 que.add(20);
 que.add(30);
 Integer ele = null;
 while( !que.isEmpty()){
     ele = que.element();
     //TODO : Processing
     que.remove();
 }
```

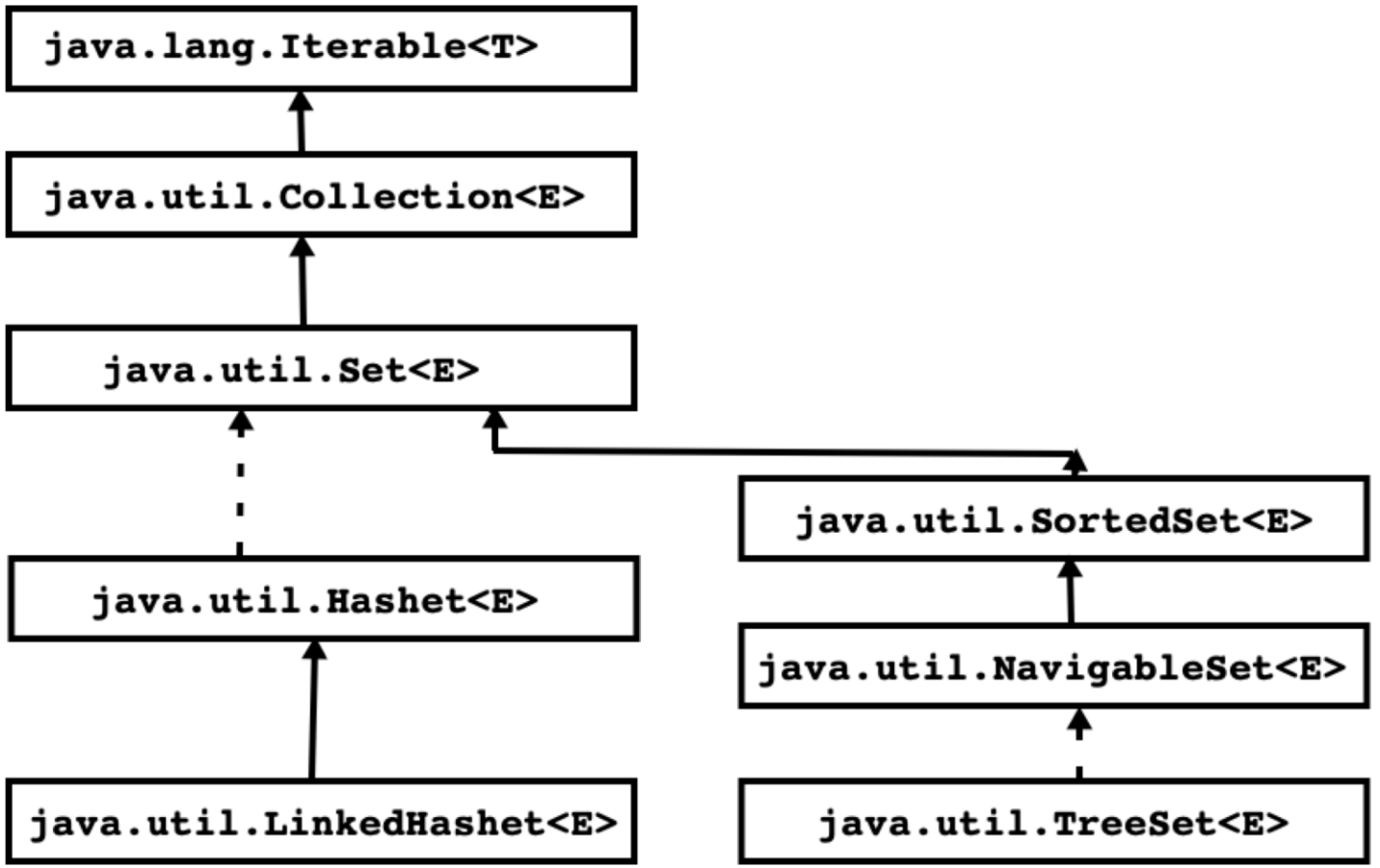
Deque<E>

- It is usually pronounced "deck".
- It is sub interface of Queue.
- It is introduced in jdk 1.6
- If we want to perform operations from bidirection then we should use Deque interface.

Summary of Deque methods

	First Element (Head)		Last Element (Tail)	
	Throws exception	*Special value*	*Throws exception*	*Special value*
Insert	`addFirst(e)`	`offerFirst(e)`	`addLast(e)`	`offerLast(e)`
Remove	`removeFirst()`	`pollFirst()`	`removeLast()`	`pollLast()`
Examine	`getFirst()`	`peekFirst()`	`getLast()`	`peekLast()`

Set Interface Hierarchy



Set<E>

- It is sub interface of java.util.Collection interface.
- HashSet, LinkedHashSet, TreeSet etc. implements Set interface. It is also called as Set collection.
- Set collections do not contain duplicate elements.
- It is introduced in jdk 1.2

TreeSet<E>

- It is Set collection.
- It can not contain duplicate element as well as null element.
- It is sorted collection.
- Its implementation is based on TreeMap
- It is unsynchronized collection.
- Using "Collections.synchronizedSortedSet()" method we can make it synchronized.
  - **SortedSet s = Collections.synchronizedSortedSet(new TreeSet(...));**
- It is introduced in jdk 1.2
- Note : If we want to manage elements of non final type inside TreeSet then non final type should implement Comparable interface.
- Instantiation
  - **Set<Integer> set = new TreeSet<>( );**

Hashing

- Hashing is a searching algorithm which is used to search element in constant time(faster searching).
- In case array, if we know index of element then we can locate it very fast.
- Hashing technique is based on "hashcode".
- Hashcode is not a reference or address of the object rather it is a logical integer number that can be generated by processing state of the object.
- Generating hashcode is a job of hash function/method.
- Generally hashcode is generated using prime number.

```
//Hash Method
private static int getHashCode(int data)
{
    int result = 1;
    final int PRIME = 31;
    result = result * data + PRIME * data;
    return result;
}
```

Hashing

- If state of object/instance is same then we will get same hashcode.
- Hashcode is required to generate slot.
- If state of objects are same then their hashcode and slot will be same.
- By processing state of two different object's , if we get same slot then it is called collision.
- Collision resolution techniques:
  - Seperate Chaining / Open Hashing
  - Open Addressing / Close Hashing
    1. Linear Probing
    2. Quadratic Probing
    3. Double Hashing / Rehashing

Hashing

- Collection(LinkedList/Tree) maintained per slot is called bucket.
- Load Factor = (Count of bucket / Total elements);
- **In hashcode based collection, if we want manage elements of non final type then reference type should override equals() and hashcode() method.**
- hashCode() is non final method of java.lang.Object class.
- Syntax:
  - public native int hashCode( );
- On the basis of state of the object, we want to generate hashcode then we should override hashCode() method in sub class.
- The hashCode method defined by class Object does return distinct integers for distinct objects. This is typically implemented by converting the internal address of the object into an integer.

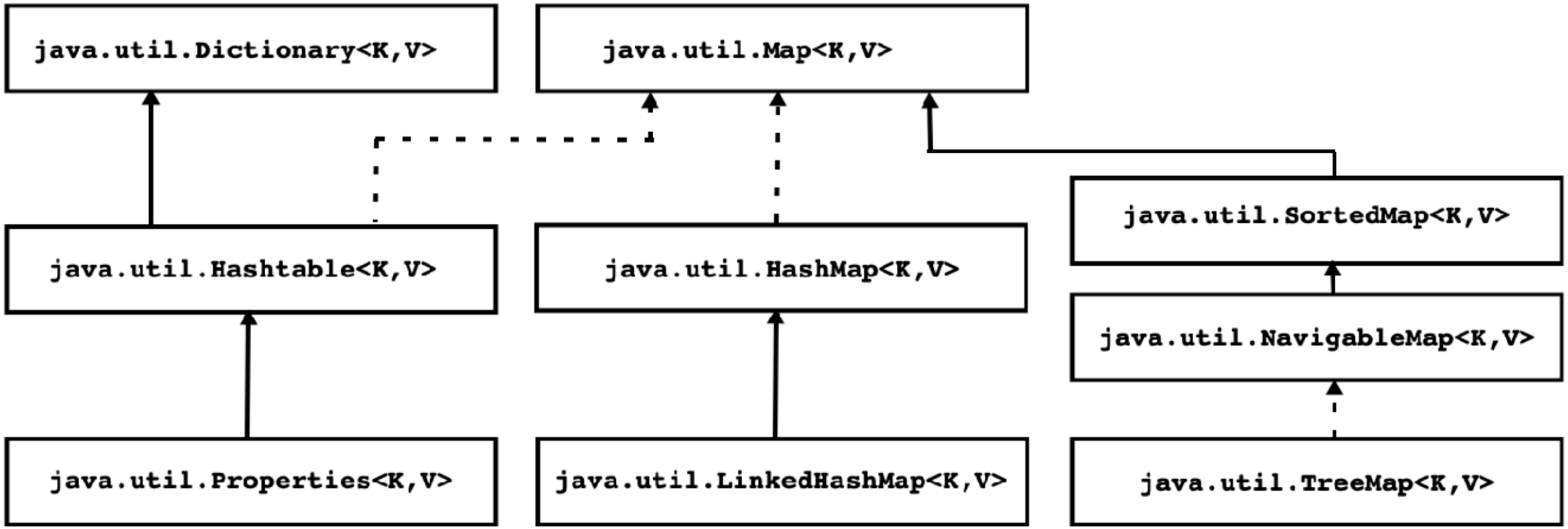
HashSet<E>

- It Set Collection.
- It can not contain duplicate elements but it can contain null element.
- It's implementation is based on HashTable.
- It is unordered collection.
- It is unsynchronized collection. Using Collections.synchronizedSet() method, we can make it synchronized.
- It is introduced in jdk 1.2
- Note : If we want to manage elements of non final type inside HashSet then non final type should override equals and hashCode() method.
- Instantiation:
  - **`Set<Integer> set = new HashSet<>();`**

LinkedHashSet<E>

- It is sub class of HashSet class.
- Its implementation is based on linked list and Hashtable.
- It is ordered collection.
- It is unsynchronized collection. Using Collections.synchronizedSet() method we can make it synchronized.
  - **Set s = Collections.synchronizedSet(new LinkedHashSet(...));**
- It is introduced in jdk 1.4
- It can not contain duplicate element but it can contain null element.

Map Interface Hierarchy



Dictionary<K,V>

- It is abstract class declared in java.util package.
- It is super class of Hashtable.
- It is used to store data in key/value pair format.
- It is not a part of collection framework
- It is introduced in jdk 1.0
- Methods:
    1. public abstract boolean isEmpty()
    2. public abstract V put(K key, V value)
    3. public abstract int size()
    4. public abstract V get(Object key)
    5. public abstract V remove(Object key)
    6. public abstract Enumeration<K> keys()
    7. public abstract Enumeration<V> elements()
- Implementation of Dictionary is Obsolete.

Map<K,V>

- It is part of collection framework but it doesn't extend Collection interface.
- This interface takes the place of the Dictionary class, which was a totally abstract class rather than an interface.
- HashMap, Hashtable, TreeMap etc are Map collection's.
- Map collection stores data in key/value pair format.
- In map we can not insert duplicate keys but we can insert duplicate values.
- It is introduced in jdk 1.2
- Map.Entry<K,V> is nested interface of Map<K,V>.
- Following are abstract methods of Map.Entry interface.

```
1. K getKey()
2. V getValue()
3. V setValue(V value)
```

Map<K,V>

- Abstract Methods of Map<K,V>
    1. boolean isEmpty()
    2. V put(K key, V value)
    3. void putAll(Map<? extends K,? extends V> m)
    4. int size()
    5. boolean containsKey(Object key)
    6. boolean containsValue(Object value)
    7. V get(Object key)
    8. V remove(Object key)
    9. void clear()
    10. Set<K> keySet()
    11. Collection<V> values()
    12. Set<Map.Entry<K,V>> entrySet()
- An instance, whose type implements Map.Entry<K,V> interface is called enrty instance.

Hashtable<K,V>

- It is Map<K,V> collection which extends Dictionary class.
- It can not contain duplicate keys but it can contain duplicate values.
- In Hashtable, Key and value can not be null.
- It is synchronized collection.
- It is introduced in jdk 1.0
- In Hashtable, if we want to use instance non final type as key then it should override equals and hashCode method.

HashMap<K,V>

- It is map collection
- It's implementation is based on Hashtable.
- It can not contain duplicate keys but it can contain duplicate values.
- In HashMap, key and value can be null.
- It is unsynchronized collection. Using Collections.synchronizedMap() method, we can make it synchronized.
  - `Map m = Collections.synchronizedMap(new HashMap(...));`
- It is introduced in jdk 1.2.
- Instantiation
  - `Map<Integer, String> map = new HashMap<>( );`
- **Note : In HashMap, if we want to use element of non final type as a key then it should override equals() and hashCode() method.**

LinkedHashMap<K,V>

- It is sub class of HashMap<K,V> class
- Its implementation is based on LinkedList and Hashtable.
- It is Map collection hence it can not contain duplicate keys but it can contain duplicate values.
- In LinkedHashMap, key and value can be null.
- It is unsynchronized collection. Using Collections.synchronizedMap() method we can make it synchronized.
  - **Map m = Collections.synchronizedMap(new LinkedHashMap(...));**
- LinkedHashMap maintains order of entries according to the key.
- Instantiation:
  - **Map<Integer, String> map = new LinkedHashMap<>();**
- It is introduced in jdk 1.4

TreeMap<K,V>

- It is map collection.
- It can not contain duplicate keys but it can contain duplicate values.
- It TreeMap, key not be null but value can be null.
- Implementation of TreeMap is based on Red-Black Tree.
- It maintains entries in sorted form according to the key.
- It is unsynchronized collection. Using Collections.synchronizedSortedMap() method, we can make it synchronized.
  - **SortedMap m = Collections.synchronizedSortedMap(new TreeMap(...));**
- Instantiation:
  - **Map<Integer, String> map = new TreeMap<>();**
- It is introduced in jdk 1.2
- Note : In TreeMap, if we want to use element of non final type as a key then it should implement Comparable interface.

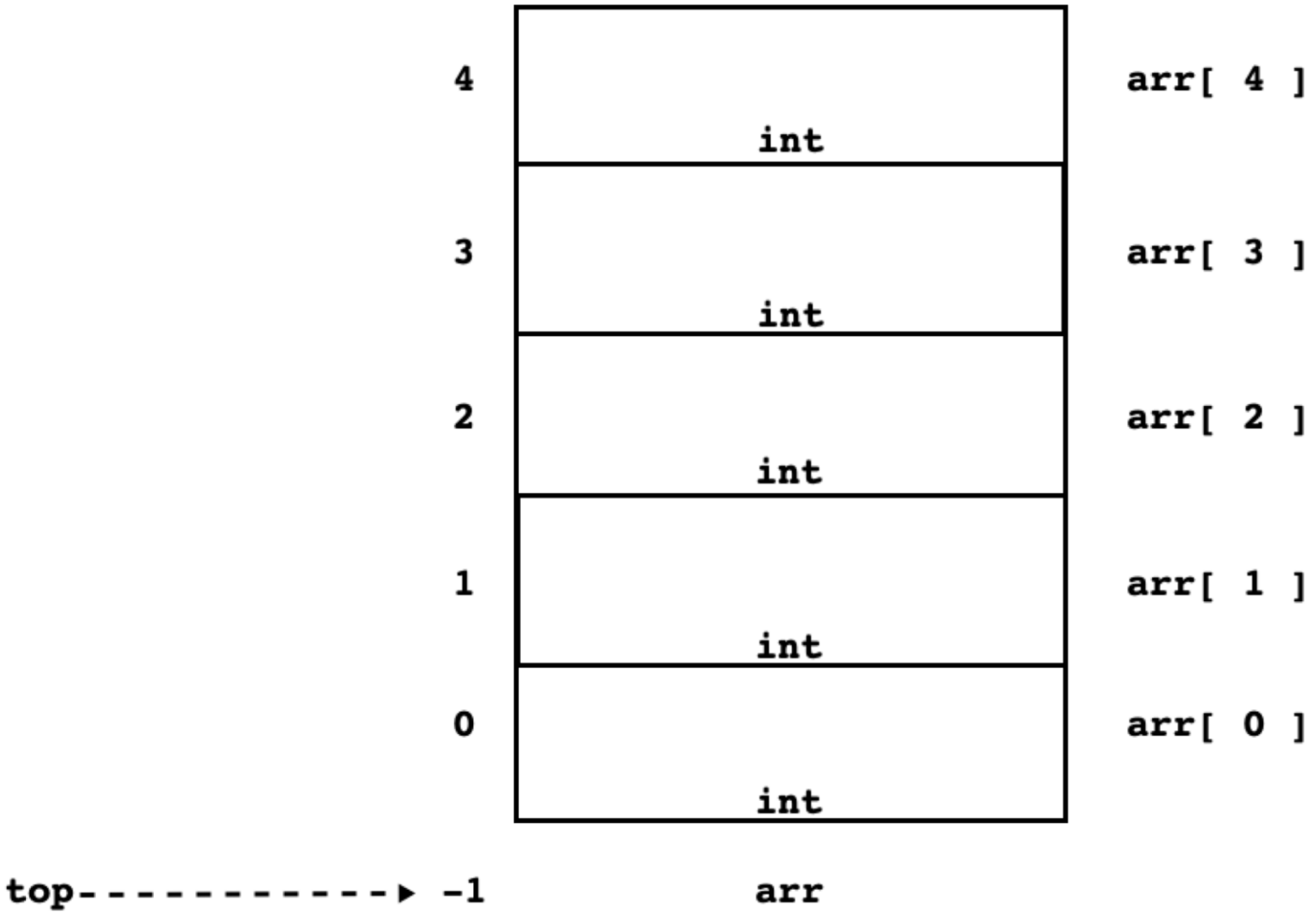
Introduction

- If we want to manage data in RAM efficiently then we should use data structure. Data structure is also called as collection.
- Types of collection:
  - Linear collection
    1. Array
    2. Stack
    3. Queue
    4. LinkedList
  - Non linear collection
    1. Tree
    2. Graph
    3. Hashtable

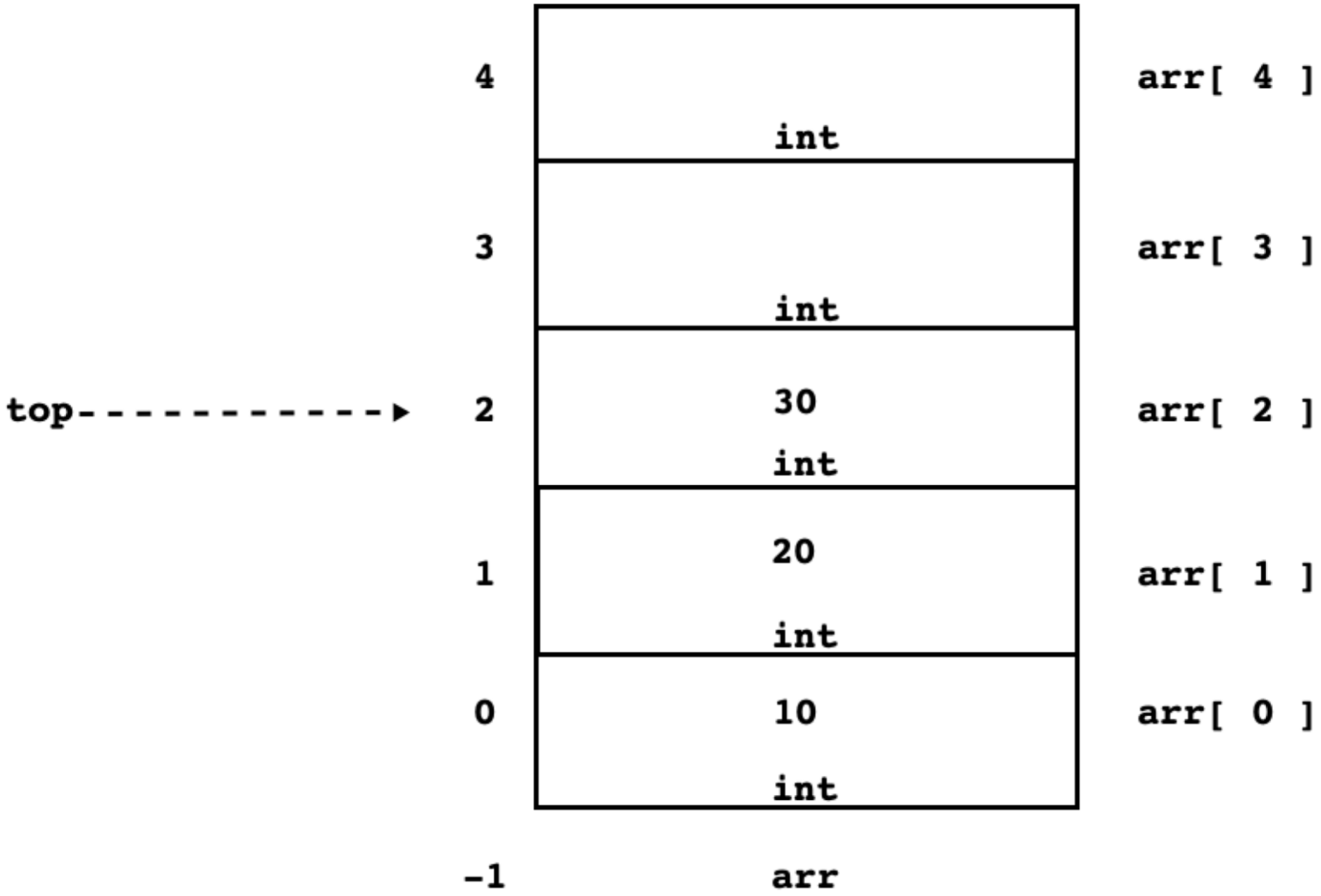
Stack

- It is a linear data structure/collection.
- In a stack, we can manage data in Last In Fast Out manner(LIFO).
- We can perform following operations on Stack:
  1. empty
  2. full
  3. push
  4. peek
  5. pop

Stack empty operation



Stack push operation

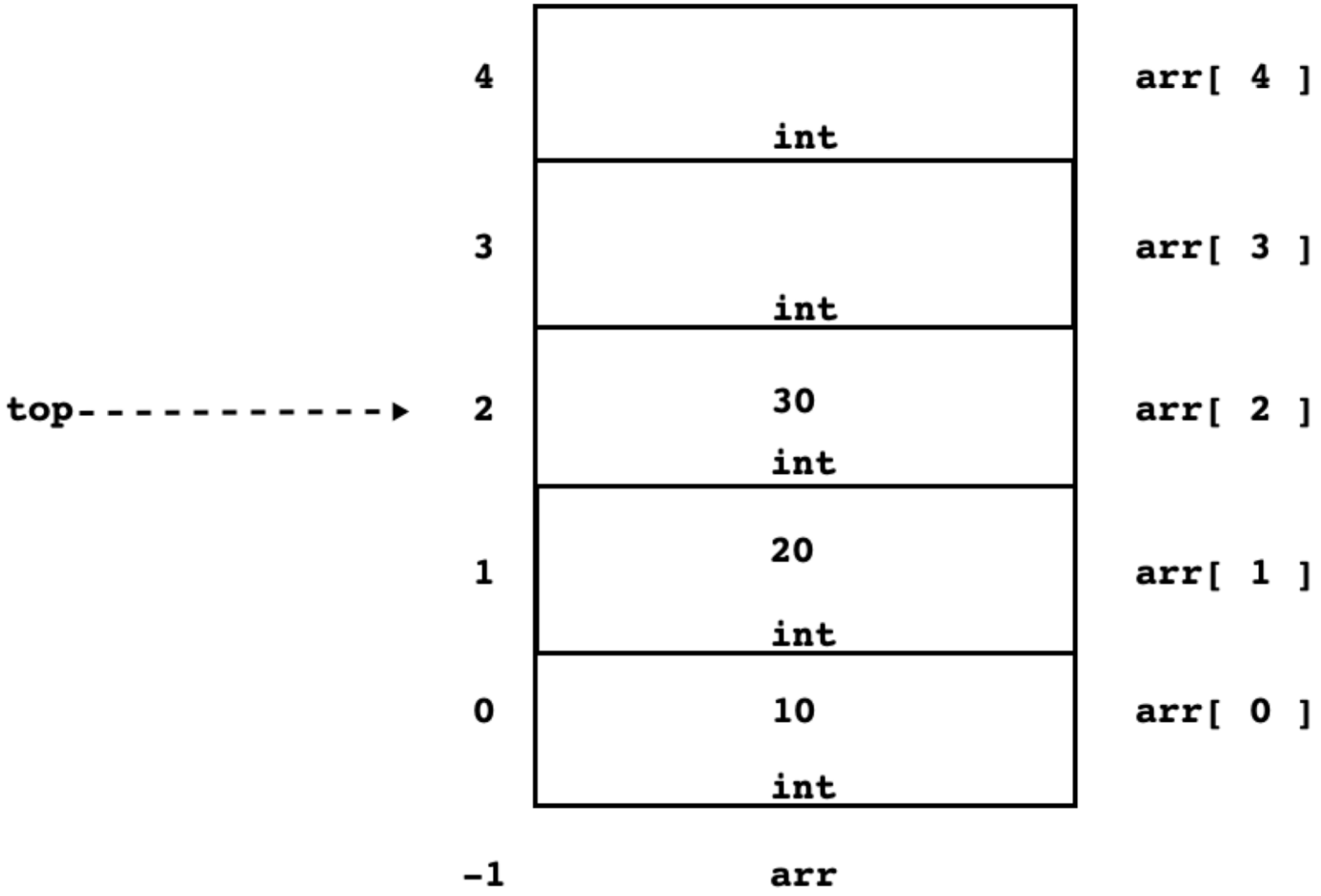



```
int data = 10;

int data = 20;

int data = 30;
top = top + 1;
arr[ top ] = data ;
```

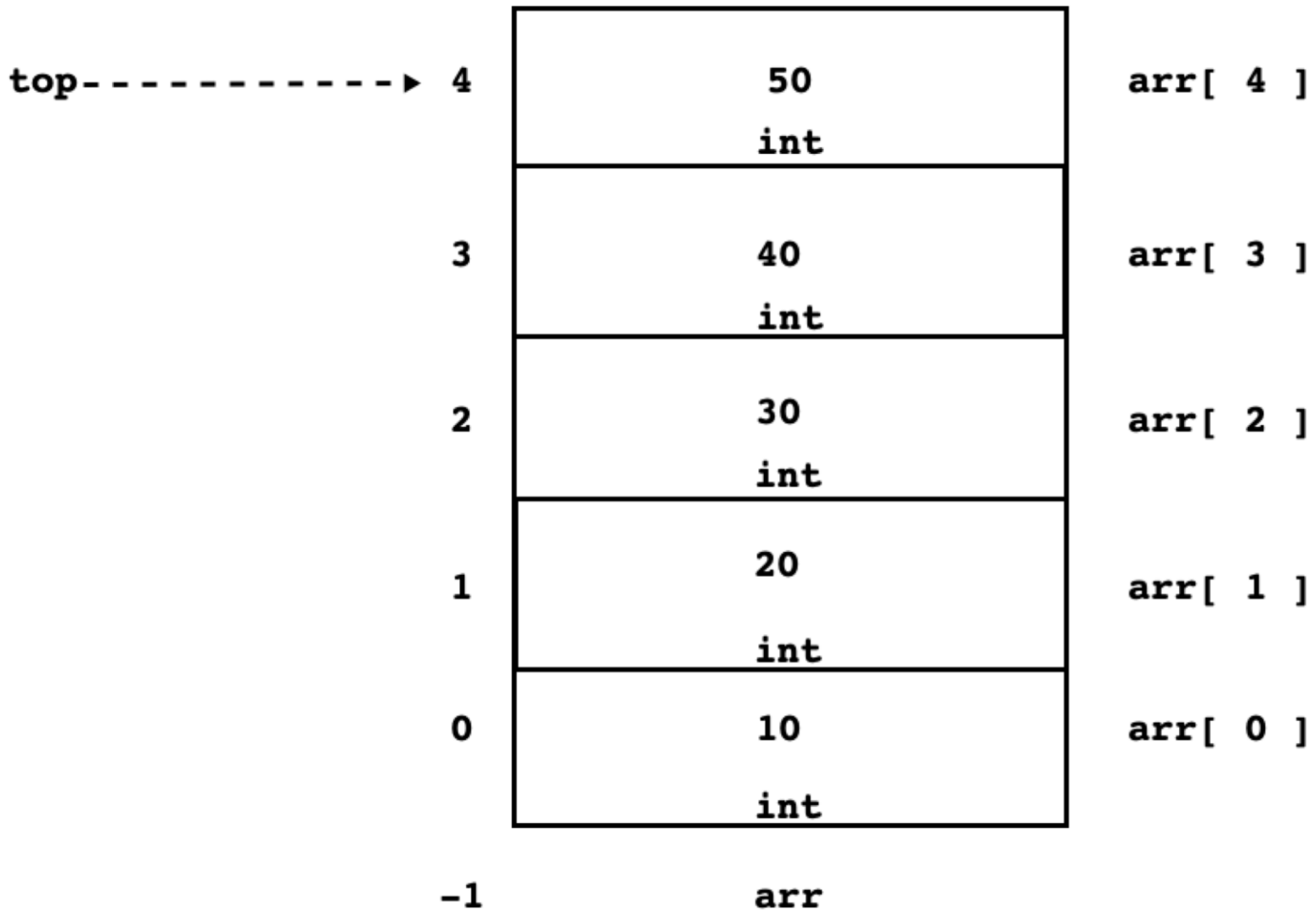
Stack peek operation



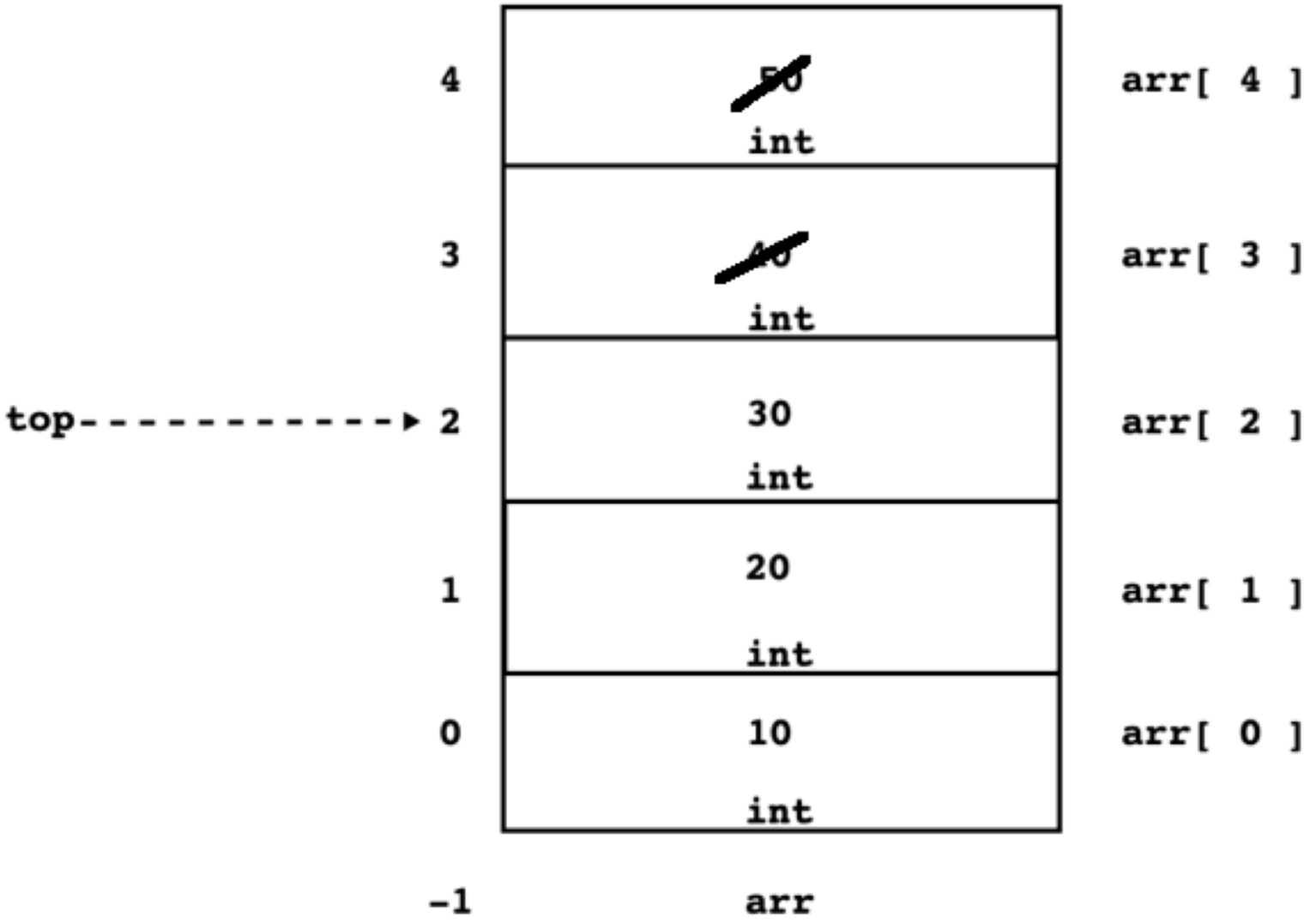

```
//top = 2;

int data = arr[ top ];
```

Stack full operation

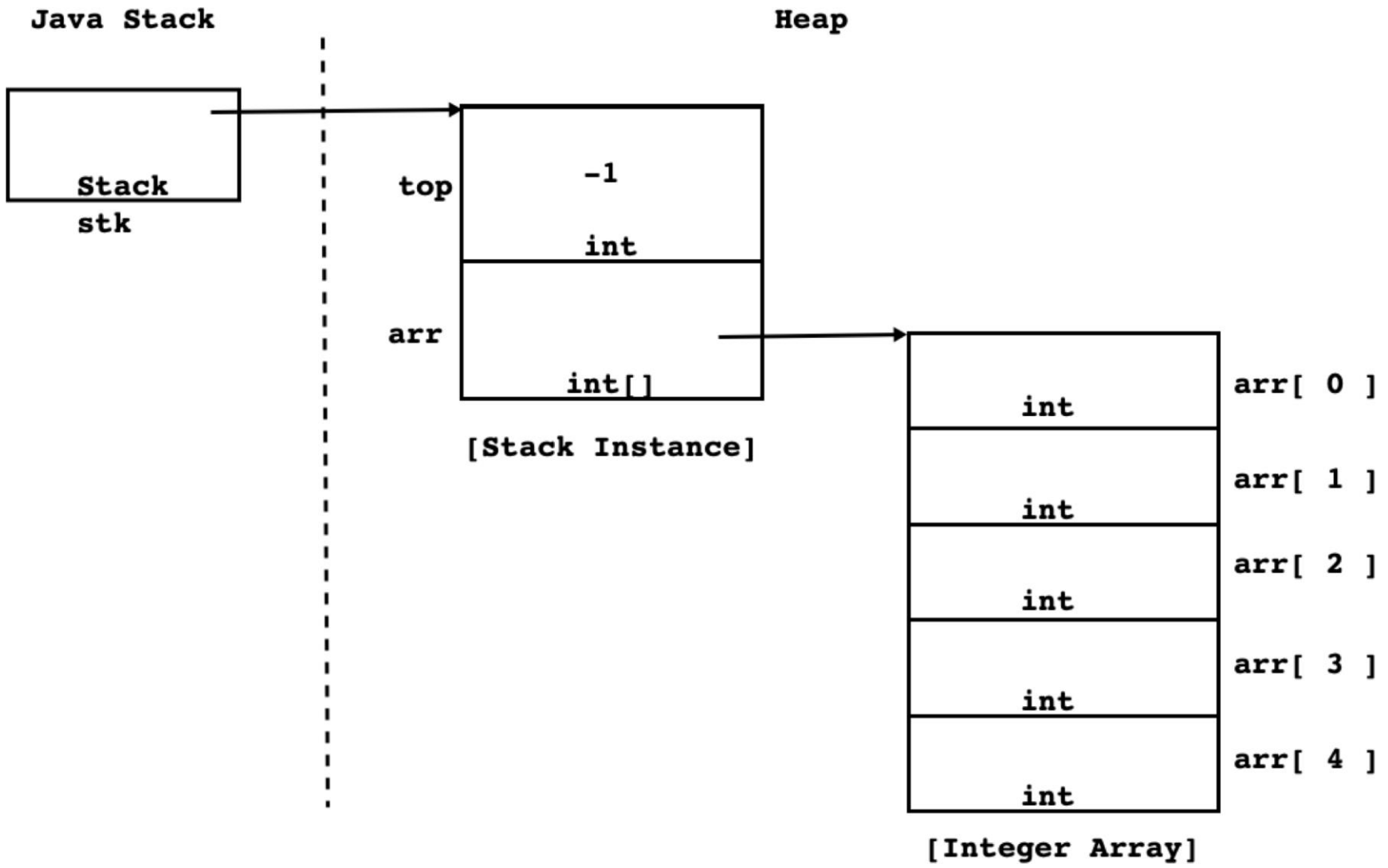


Stack pop operation




```
– – this.top;
```

Stack Simulation



Thank You.

[akshita.chanchlani@sunbeaminfo.com]