Java Fundamentals

Generics & Collections

Topics

- Generics
- Comparable & Comparator
- Collections
- Immutable objects
- Objects lifecycle

Generics

Generics

What is generics?

- Java Generic methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods, or with a single class declaration, a set of related types, respectively.
- Using Java Generic concept, we might write a generic method for sorting an array of objects, then invoke the generic method with Integer arrays, Double arrays, String arrays and so on, to sort the array elements.

Generics

Generic methods

- You can write a single generic method declaration that can be called with arguments of different types.
- Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately.
- All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type (< E >).
- Each type parameter section contains one or more type parameters separated by commas.
- A type parameter, also known as a type variable, is an identifier that specifies a generic type name.
- The type parameters can be used to declare the return type and act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments.
- A generic method's body is declared like that of any other method.
- Type parameters can represent only reference types, not primitive types.

Generics

Bounded types

- There may be times when you'll want to restrict the kinds of types that are allowed to be passed to a type parameter. For example, a method that operates on numbers might only want to accept instances of Number or its subclasses. This is what bounded type parameters are for.

- To declare a bounded type parameter, list the type parameter's name, followed by the **extends** keyword, followed by its upper bound.

Generics
Wildcards

- “?”- used to refer to an **unknown** type.
- Generic wildcards are used when we want to use a generic class as a parameter in a method and we don’t want to limit the type
- Object is the supertype of all Java classes, but, a collection of Object is not the supertype of any collection. (e.g.: List<Object> is not the supertype of List<String> => assigning a variable of type List<Object> to a variable of type List<String> will cause a compiler error)
- Wildcards can be used with both upper bounding (**extends**) and lower bounding(**super**)
- More here

Generics

Generic classes

- A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section.
- As with generic methods, the type parameter section of a generic class can have one or more type parameters separated by commas.
- These classes are known as parameterised classes or parameterised types because they accept one or more parameters.

- More about generics here, here and here.

Comparable & Comparator

Comparable & Comparator

Comparable

- A **comparable object** can compare itself with another object.
- The class itself must implement the **java.lang.Comparable** interface. This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's natural ordering, and the class's compareTo method is referred to as its natural comparison method.
- *public int compareTo(Object obj)*: It is used to compare the current object with the specified object. It returns:
  - positive integer, if the current object is greater than the specified object.
  - negative integer, if the current object is less than the specified object.
  - zero, if the current object is equal to the specified object.

Comparable & Comparator

Comparator

- Unlike Comparable, a **Comparator** is external to the element type we are comparing.
- It's a separate class. We create multiple separate classes (that implement **java.util.Comparator**) to compare by different members.
- To compare objects using a comparator, we need to:
  - Create a class that implements Comparator (and the compare() method that does the work previously done by compareTo()).
  - The *compare()* method in Java compares two class specific objects (x, y) given as parameters. It returns 0 (if x == y), 1 (if x > y) or -1 (if x < y)
- More on comparable and comparator here and here

Collections

Collections

Collections Framework

- package: j**ava.util**
- The collections framework contains:
  - Interfaces
  - Implementations
  - Algorithms
- Collections offer implementations for:
  - List (list of elements)
  - Map (key-value pairs)
  - Set (unique elements)
- **Collection** interface – implemented by most of the collections
- **Collections** class – static methods useful for working with collections

Collections

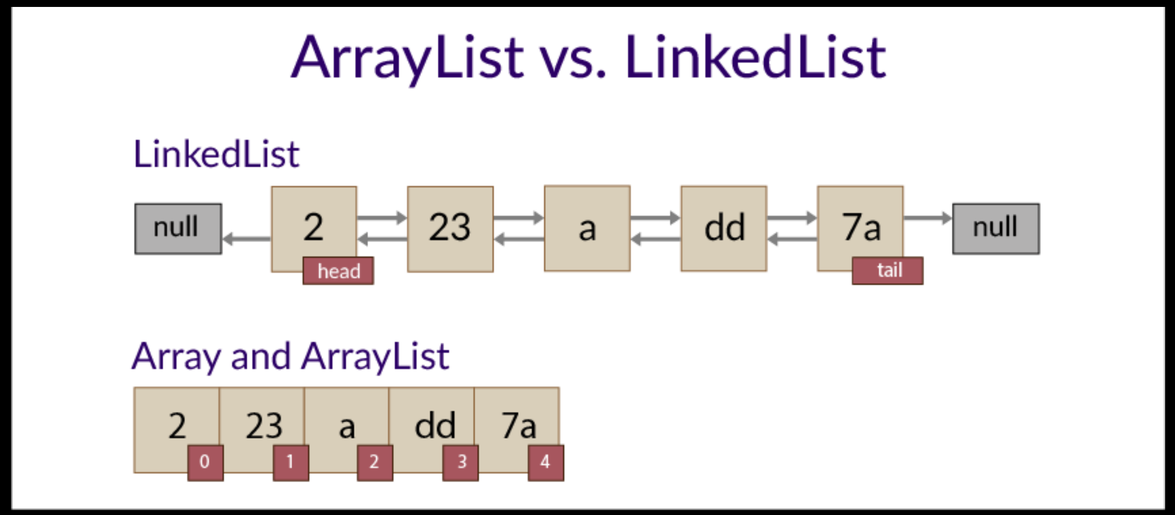
Iterators

- We can iterate over a collection:
  - Using a for-each loop
  - Using an Iterator:
    - .iterator() method to obtain it
    - Iterator interface
    - using an Iterator, we can delete elements at the same time (not possible with for-each loop)

Collections

List

- data structure: array, dynamic list
- **ListIterator** – interface that extends Iterator and provides reverse order iteration



- Implementations:
  - **ArrayList** – good for reads
  - **LinkedList** – good for insert / remove from linked list

Collections

Set

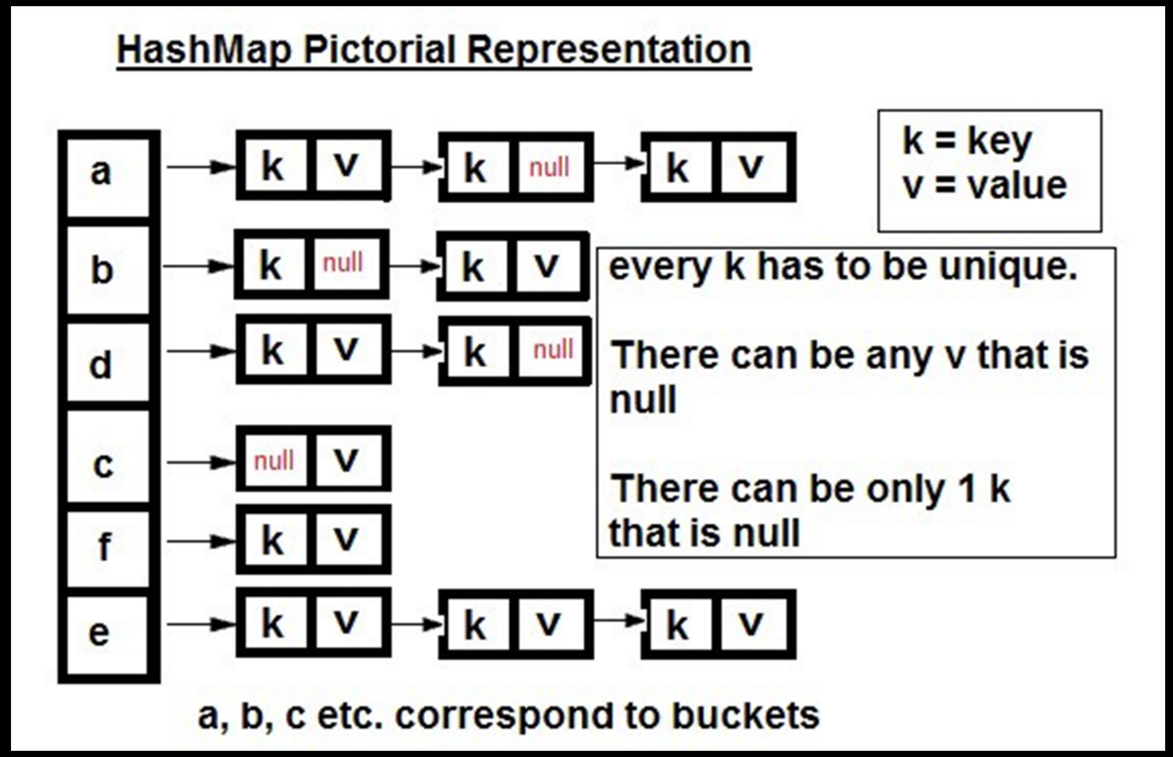
- unique elements
- Implementations:
  - **HashSet**:
    - uses a hash table
    - does not guarantee order
  - **LinkedHashSet**
    - uses a hash table
    - guarantees the insertion order
  - **TreeSet**
    - uses a red-black tree
    - keeps the elements sorted

Collections

Map

- Key – value pairs (keys must be unique)
- **Map.Entry** interface – models the key-value pair

- Implementations:
  - **HashMap**:
    - uses a hash table
    - does not guarantee order
  - **LinkedHashMap**
    - uses a hash table
    - guarantees the insertion order
  - **TreeMap**
    - uses a red-black tree
    - keeps the elements sorted



Collections

Map - equals and hashcode contract

- If we override **equals()** -> we must also override **hashCode()**
- If we use a hash-based collection, we must override both of them

- **Contract**:
  - If we call hashCode() multiple times on the same object -> same result
  - If obj1.equals(obj2) -> hashcode obj1 == hashcode obj2
  - If !obj1.equals(obj2) -> hashcodes don’t have to be different, but it’s recommended.
- More about collections here , here and here.

Immutable objects

Immutable objects

- object whose internal state remains **constant** after it has been entirely created (eg: String class)
- Strategy to create immutable objects:
  - Don’t provide “setter” methods - methods that change the fields
  - Make all fields private and final
  - Don’t allow subclasses to override methods (eg: make the class final)
  - If the class includes references to other mutable objects, don’t allow them to be changed
    - Don’t provide methods that modify them
    - Don’t store references of those mutable classes, create copies and store the copies instead
- More here

Object lifecycle

Objects lifecycles

Garbage collector

- tracks every object available in the JVM heap space, and removes the unused ones
- Mark - identifies which parts of the memory are in use and which aren’t
- Sweep - removes the elements that were marked for de-allocation

Objects lifecycle

Garbage collector

- Advantages:
  - No manual memory allocation or deallocation
- Disadvantages:
  - More CPU power needed (to keep track of all the objects) => lower performance
  - Programmers have no control over when the objects are de-allocated from memory

Objects lifecycle

Garbage collector

- Implementations in JVM:
  - Serial Garbage Collector- simplest GC implementation, works with a single thread. It freezes all application threads when it runs (not ideal for multi threaded applications)
  - Parallel Garbage Collector - default (Throughput Collectors); uses multiple threads
  - CMS Garbage Collector - deprecated since Java 9
  - G1 Garbage Collector
  - More here

Object lifecycle

Object states

1. Java Object lifecycle states:
2. Created
3. In use
4. Invisible
5. Unreachable
6. Collected
7. Finalised
8. De-alocated

Objects lifecycle

Created

- New memory is **allocated** for it
- If it was assigned to a variable, it moves into “in use” status

MyClass myClass = new MyClass()

Objects lifecycle

In use

- If there is at least one **strong reference** to the object, it is considered in use

MyClass myClass = new MyClass();

myClass.setField(“Field)

Objects lifecycle

Invisible

- An object moves to “Invisible” state when there are no longer any strong references that are accessible to the program, even though **there might still be references**

MyClass myClass = new MyClass();

myClass.setField(“Field)

myClass = null;

- Now, myClass object is available to be deleted by Garbage Collector, but it will be collected only if JVM needs memory.

Objects lifecycle

Unreachable

- An object enters an “unreachable” state when **no more strong references to it exist**. When an object is unreachable then it is a state for collection.

Objects lifecycle

Collected

- the garbage collector has **recognised** an object as unreachable and readies it for final processing as a precursor to de-allocation. If the object has a finalize method, then it is marked for finalization.

Objects lifecycle

Finalised

- An object is in the “finalized” state if it is still unreachable after it’s finalize method has been run.
  - **finalize()** method of Object class is a method that the Garbage Collector always calls just before the deletion/destroying the object
  - It performs clean-up activities on that object (eg: closing files, avoiding resource leaks)
- A finalized object is **awaiting de-allocation**.
- Deprecated since Java9, will be removed soon
- More on finalize() here and here

Objects lifecycle

De-allocation

- final step in garbage collection.
- If the object is thill unreachable, then it is de-allocated, memory is released