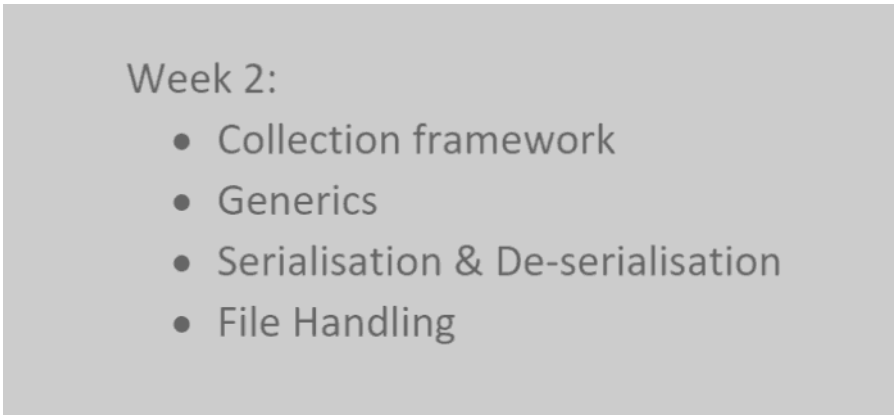
Week 2

Monday, March 27, 2023 1:28 PM



https://www.scaler.com/topics/java/collections-in-java/

Collections :

Collections is a group of objects of same type which is represented as a single unit.

Collections framework :

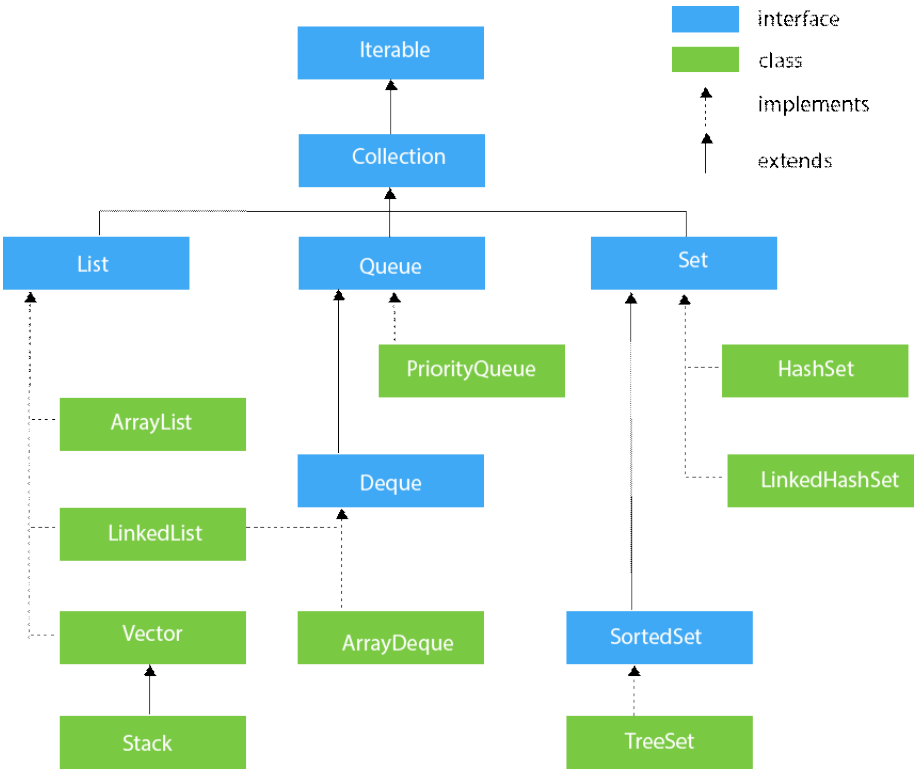
Collection framework provides a set of classes and interfaces which can be used to represent a group of objects.
The Collections Framework is defined as a unified architecture for representing and manipulating collections. In Java, the Collections Framework is a hierarchy of interfaces and classes that provides an easy management of a group of objects.

Why Collections framework?

As we can observe, none of these collections(Array, Vector, or Hashtable) implements a standard member access interface, it was very difficult for programmers to write algorithms that can work for all kinds of Collections. Another drawback is that most of the 'Vector' methods are final, meaning we cannot extend the 'Vector' class to implement a similar kind of Collection. Therefore, Java developers decided to come up with a common interface to deal with the above-mentioned problems and introduced the Collection Framework in JDK 1.2 post which both, legacy Vectors and Hashtables were modified to conform to the Collection Framework.

But prior to the Collections, Vectors, Stacks and Arrays were there. They had one major disadvantage, and that was lesser similarities. They didn't have a common interface and interconnection with each other. In this case, it becomes tedious for the user to remember all the functions and syntax. Plus, the conventional ways like arrays and stacks weren't providing the desired performance and flexibility.

Collections Hierarchy :



Advantages of Collections :

1. **Provides high-performance and high efficiency**. This is due to the fact that various implementations of each interface are interchangeable, so programs can be written by switching implementations.
2. **Some methods of each interface of the Collections Framework have a uniform implementation**. The Collections API has basic interfaces like Set, Map and List, so the classes that implement them have a few common set of methods.
3. **Reduces the programming effort by providing useful data structures and algorithms**. We don't have to write our own data structure as it has already been provided to us.
4. **Facilitates code reusability**. Collections Framework provides common classes and interfaces that can be used with different types of collections.
5. **Provides uniform interface.**

Iterable vs Iterator :

Iterable	Iterator
Represents a collection that can be iterated over using a *for*-each loop	Represents an interface that can be used to iterate over a collection
When implementing an *Iterable*, we need to override the *iterator()* method	When implementing an *Iterator*, we need to override the *hasNext()* and *next()* methods
Doesn't store the iteration state	Stores the iteration state
Removing elements during the iteration isn't allowed	Removing elements during the iteration is allowed

```
public Iterator<E> iterator(){
return new Itr();
}
```

Objects of Iterable can be Iterated in three ways:

1. Enhanced for loop(for-each loop)
2. Iterable forEach loop
3. Iterator<T> interface

List Collection :

The list interface extends the collection interface. A list is used to store ordered collection of data and it may contain duplicates. Ordered collection means the order in which the elements are being inserted and they contain a specific value. The elements present, can be accessed or inserted by their position in the list using zero-based indexing. The list interface is **implemented by LinkedList, ArrayList, Vectors and Stack classes**.

1) ArrayList :

a. So, you must've figured it out from the name itself, that ArrayList is similar to Arrays. They are also called as dynamic arrays. That means, it does not have a fixed size. Its size can be increased or decreased if elements are added or removed.
b. It implements the List Interface.
c. It is similar to Vectors in C++.
d. Since the ArrayList cannot be used for primitive data types like int,char etc. , we need to use a wrapper class.
e. Resizing and Continuous memory is difficult in ArrayList.

Capacity of an ArrayList

Technically, the default capacity (DEFAULT_CAPACITY) of a newly created ArrayList is 10. However, Java 8 changed how this initial capacity is used for performance reasons.
It's not used immediately and is guaranteed lazily once a new item is added to the list.
So, the default capacity of an empty ArrayList is 0 and not 10 in Java 8. Once the first item is added, the DEFAULT_CAPACITY which is 10 is then used.
Since there is no built-in method to get the default capacity, let's use reflection to return it.

2) Linked List :

a. The LinkedList class extends the AbstractSequentialList and it also extends the List, Deque and Queue interface.
b. By this, we get a linked-list data structure.
c. Linked List is a linear data structure where the elements are called as nodes.
d. Here, each node has two fields- data and next. Data stores the actual piece of information and next points to the next node. 'Next' field is actually the address of the next node.
e. Elements are not stored in a contiguous memory, so direct access to that element is not possible.
f. LinkedList uses Doubly Linked List to store its elements while ArrayList internally uses a dynamic array to store its elements. LinkedList is faster in manipulation of data as it is node based which makes it unique.
g. LinkedList is non-synchronised means multiple threads at a time can access the code. This means if one thread is working on LinkedList, other threads can also get a hold of it. Multiple operations on LinkedList can be performed at a time. For example, if addition is being performed by one thread, other operation can be performed by some other thread too.

Data growth :
Internally, both the ArrayList and Vector hold onto their contents using an Array. When an element is inserted into an ArrayList or a Vector, the object will need to expand its internal array if it runs out of room. A Vector defaults to doubling the size of its array, while the ArrayList increases its array size by 50 percent.

Linked lists are preferable over arrays when:

1. you need constant-time insertions/deletions from the list (such as in real-time computing where time predictability is absolutely critical)
2. you don't know how many items will be in the list. With arrays, you may need to re-declare and copy memory if the array grows too big
3. you don't need random access to any elements
4. you want to be able to insert items in the middle of the list (such as a priority queue)

Arrays are preferable when:

1. you need indexed/random access to elements
2. you know the number of elements in the array ahead of time so that you can allocate the correct amount of memory for the array
3. you need speed when iterating through all the elements in sequence. You can use pointer math on the array to access each element, whereas you need to lookup the node based on the pointer for each element in linked list, which may result in page faults which may result in performance hits.
4. memory is a concern. Filled arrays take up less memory than linked lists. Each element in the array is just the data. Each linked list node requires the data as well as one (or more) pointers to the other elements in the linked list.

3) Vector :

a. Like ArrayList, Vectors in Java are used for dynamic arrays.
b. It extends the AbstractList and implements the List interface.
c. Vector is synchronized. Synchronized means only one thread at a time can access the code. This means if one thread is working on Vector, no other thread can get a hold of it. Only one operation on vector can be performed at a time. For example, if addition is being performed by one thread, other operation cannot be performed until the first one is over.

While initializing vectors in Java, one should keep in mind some exceptions that can occur:

a. IllegalArgumentException: This is thrown if the initial size of the vector defined is negative.
b. NullPointerException: This is thrown if the specified collection passed in the constructor is null.

3) TreeMap :
TreeMap sorts elements in natural order and doesn't allow null keys because compareTo() method throws NullPointerException if compared with null.

Set allows null values as it internally uses HashMap

ArrayDeque :

In this tutorial, we'll show how to use Java's ArrayDeque class – which is an implementation of the Deque interface.
An ArrayDeque (also known as an "Array Double Ended Queue", pronounced as "ArrayDeck") is a special kind of a growable array that allows us to add or remove an element from both sides.
An ArrayDeque implementation can be used as a Stack (Last-In-First-Out) or a Queue(First-In-First-Out).

The API at a Glance
For each operation, we basically have two options.
The first group consists of methods that throw an exception if the operation fails. The other group returns a status or a value:

Operation	Method	Method throwing Exception
Insertion from Head	*offerFirst(e)*	*addFirst(e)*
Removal from Head	*pollFirst()*	*removeFirst()*
Retrieval from Head	*peekFirst()*	*getFirst()*
Insertion from Tail	*offerLast(e)*	*addLast(e)*
Removal from Tail	*pollLast()*	*removeLast()*
Retrieval from Tail	*peekLast()*	*getLast()*

CopyOnWriteArrayList :

https://www.baeldung.com/java-copy-on-write-arraylist
This is a very useful construct in the multi-threaded programs – when we want to iterate over a list in a thread-safe way without an explicit synchronization.
The CopyOnWriteArrayList was created to allow for the possibility of safe iterating over elements even when the underlying list gets modified.
Because of the copying mechanism, the remove() operation on the returned Iterator is not permitted – resulting with UnsupportedOperationException.

HashMap :

https://www.geeksforgeeks.org/internal-working-of-hashmap-java/
https://www.geeksforgeeks.org/load-factor-and-rehashing/

ConcurrentHashMap :

If we try to modify the collection while iterating over it, we get ConcurrentModificationException. Java 1.5 introduced Concurrent classes in the java.util.concurrent package to overcome this scenario. ConcurrentHashMap is the Map implementation that allows us to modify the Map while iteration. The ConcurrentHashMap operations are thread-safe. ConcurrentHashMap doesn't allow null for keys and values.
https://www.digitalocean.com/community/tutorials/concurrenthashmap-in-java

Time Complexity :

Collection	Add	Remove	Search	Data Structure	Default Size	Resizing	Synchronized
ArrayList	O(1) Amortized	O(N)	O(N)	Dynamic Array	10 (0)	50% of old capacity	No
LinkedList	O(1)	O(N)	O(N)	Linked List	No fixed size (Not contiguous)	Increases and decreases as needed	No
ArrayDeque				Array Double Ended Queue (Circular array)	16	Doubles	No
PriorityQueue	O(log N)	O(log N)	O(N)	Priority queue (Internally uses Array)	11	Not specified	No
HashMap	O(1)	O(1)	O(1)	Hash Table	16 (Array size)	Doubles	No

TreeMap							
LinkedHashMap							
HashSet	O(1)	O(1)	O(1)	Hash Table			
LinkedHashSet							
TreeSet							
Vector	O(1) Amortized	O(N)	O(N)	Dynamic Array		Doubles	Yes (One operation at a time)
Stack	O(1) (push())	O(1) (pop())	O(N)				
Concurrent HashMap							
CopyOnWriteArrayList							
HashTable							

Comparator vs Comparable :

Use **Comparable** when:

1) The class has a natural ordering that is inherent to the class.
2) The ordering is consistent with the equals method.
3) The class is not likely to change the ordering in the future.

For example, the String class implements Comparable, because it has a natural ordering based on lexicographic order.

Use **Comparator** when:

1) The class does not have a natural ordering, or the ordering is inconsistent with the equals method.
2) You need to sort the class in multiple ways.
3) You need to sort objects of a class that you do not control or cannot modify.

Generics in Java :

https://www.digitalocean.com/community/tutorials/java-generics-example-method-class-interface

Serialization & Deserialization :

https://www.scaler.com/topics/java/serialization-and-deserialization/

Serialization :

```
// Serialization
try {
        Employee emp = new Employee("James",132 );

        FileOutputStream fout= new FileOutputStream("f.txt");

        ObjectOutputStream out= new ObjectOutputStream(fout);

        out.writeObject(emp);
        out.flush();
        out.close();
        System.out.println("Serialization successful");
}
```

Important Points :

There are a few points you need to remember while implementing serialization.

1. Serializable Interface must be implemented by all the associated objects.
2. If the parent class has already implemented the Serializable interface, in that case, a child class doesn't have to implement it.
3. During the serialization process, only **non-static** data members are saved.
4. While converting the byte stream back to the original object, the constructor of the object is not called.
5. Serializable Interface has no methods or fields of its own.

Advantages of Serialization :

Advantages of Serialization

Mostly we have covered the merits and use cases of Serialization in the section where we discussed its need. But to summarize, the main advantages of serialization are:

1. It helps to preserve the state or the data of the object.
2. It is platform-independent.
3. It makes a time-saving and efficient transfer of objects happen between two platforms.
4. It helps in creating replicas of objects, **, i.e., cloning them**.
5. It is easy to understand and customizable.
6. It allows encrypted and safe Java computing.

Serial Version UID :
We use the serialVersionUID attribute to remember versions of a Serializable class to verify that a loaded class and the serialized object are compatible.
SerialVersionUID is a unique identifier for each class, JVM uses it to compare the versions of the class ensuring that the same class was used during Serialization is loaded during Deserialization.

The serialVersionUID attributes of different classes are independent. Therefore, it is not necessary for different classes to have unique values.

If the Serial Version UID of the loaded class does not match with the serialized object's ID, it throws InvalidClassException.

If we don't define a serialVersionUID state for a Serializable class, then Java will define one based on some properties of the class itself such as the class name, instance fields, and so on.

3. Compatible Changes

Let's say we need to add a new field *lightningPort* to our existing *AppleProduct* class:

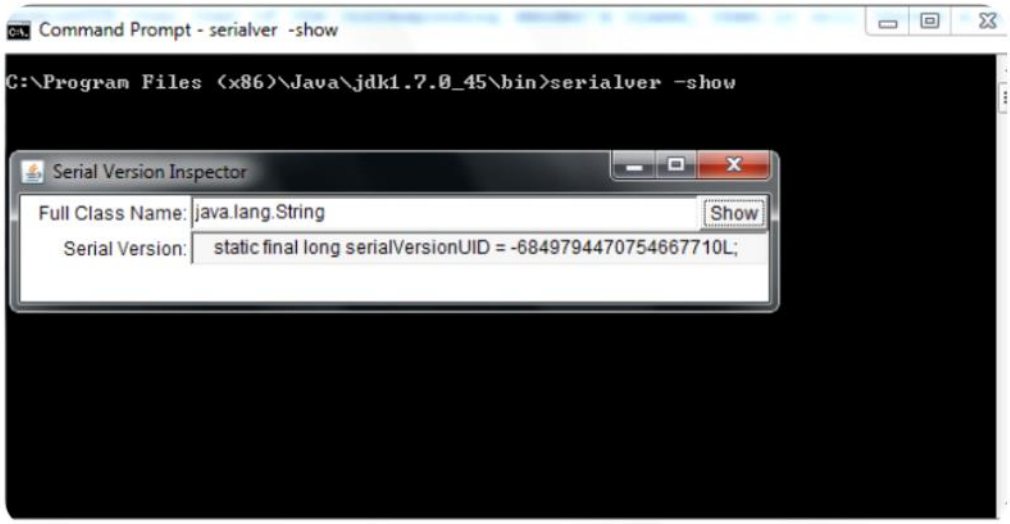
```
public class AppleProduct implements Serializable {
//...
    public String lightningPort;
}
```

Since we are just adding a new field, **no change in the *serialVersionUID* will be required**. This is because, **during the deserialization process, *null* will be assigned as the default value for the *lightningPort* field**.

Serialver Command in Java :

In order to get the Serial version UID for a class, the Java Development Kit has a built-in command, serialver. The command is written as : serialver [-classpath classpath] [-show] [classname...]
So we have two options here,

-classpath: This option helps us specify the path of the class or where to look for a particular class.
-show: show option displays a simple user interface, where you can enter the full class name and press ENTER or SHOW to get the serial version UID.



Deserialization :

```java
// Deserialization
try {
        FileInputStream fin = new FileInputStream("f.txt");

        ObjectInputStream in = new ObjectInputStream(fin);

        emp1 = (Demo)in.readObject();
        in.close();

        System.out.println("Deserialization successful");

        System.out.println( "Name: " + emp1.name);

        System.out.println( "id: " + emp1.id);
}
```

Java Deserialize Vulnerabilities :
Java deserialize vulnerabilities are security vulnerabilities that occur when undesired or modified objects are inserted during the process of serialization-deserialization by malicious activities. Let's consider that for your Java application. You're reconstructing the object from the byte stream. So you're expecting the already serialized object, let's say obj1. However, instead of obj1, you get obj2. The retrieved object is thus a result of some malicious activities; this is a Java deserialize vulnerability. Untrusted and malicious byte-streams can easily exploit vulnerable deserialization code.

Prevent Deserialize Vulnerabilities :

- The most basic approach is performing inspection of the objects from a deserialized object stream or in other words, basic filtration of the ObjectInputStream. There are several libraries to perform these validation actions.
- The other way is to forbid objects of some classes from being deserialized, the **blacklisting** approach.
- We can also allow a set of objects of approved classes to get deserialized, the whitelist approach. Deserialization will occur in a restrictive manner thus avoiding the chances of **deserialize vulnerabilities.**
- Keep the open source libraries up to date.

Transient keyword :
Let's consider a case where you are serializing an object and want that a certain field of the object

doesn't get serialized. To achieve this, we can take the help of the transient keyword. So, the Java transient keyword helps to prevent a particular field from being serialized.

```
import java.io.Serializable;
public class Employee implements Serializable{
        private static final long serialVersionUID =
        123L;
        String name;
        int id;
        transient int age;
}
```

Externalizable interface :
The main goal of Externalizable interface is to facilitate custom serialization and deserialization.

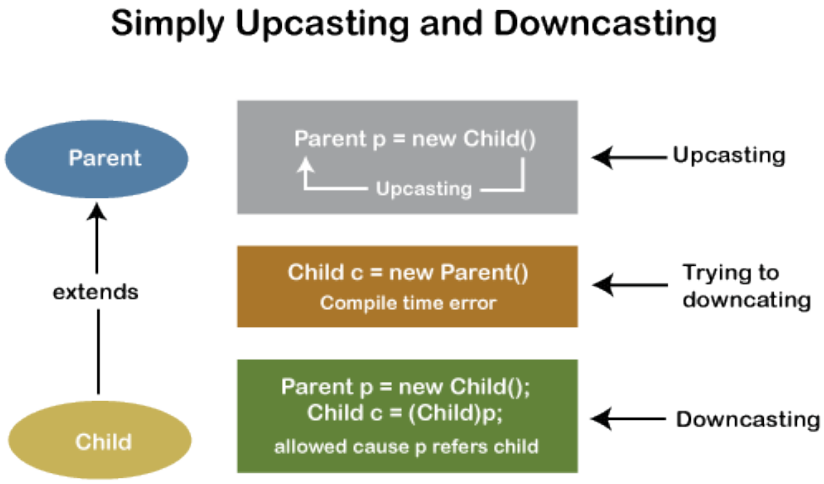
Externalizable extends from the java.io.Serializable marker interface. Any class that implements Externalizable interface should override the writeExternal(), readExternal() methods. That way we can change the JVM's default serialization behavior.

```
public class Country implements Externalizable {

    private static final long serialVersionUID = 1L;

    private String name;
    private int code;

    // getters, setters

    @Override
    public void writeExternal(ObjectOutput out) throws IOException {
        out.writeUTF(name);
        out.writeInt(code);
    }

    @Override
    public void readExternal(ObjectInput in)
      throws IOException, ClassNotFoundException {
        this.name = in.readUTF();
        this.code = in.readInt();
    }
}
```

In the **writeExternal()** method, we're adding the object's properties to the ObjectOutput stream. This has standard methods like writeUTF() for String and writeInt() for the int values.

Next, for deserializing the object, in the **readExternal()** method, we're reading from the ObjectInput stream using the readUTF(), readInt() methods to read the properties in the same exact order in which they were written.

Typecasting :



File Handling :

1) InputStreamReader and OutputStreamWriter - these classes convert byte streams to character streams and vice versa.
2) FileReader and FileWriter - these classes are used to read and write character streams, and they are built on top of InputStreamReader and OutputStreamWriter.
3) FileInputStream and FileOutputStream - these classes are used to read and write byte streams.
4) BufferedInputStream and BufferedOutputStream - these classes are used to improve the performance of reading and writing byte streams by buffering the data.
5) Buffered Reader and BufferedWriter - these classes are used to improve the performance of reading and writing character streams by buffering the data