Exception Handling or Declarations

Exceptions

When an exceptional condition occurs in the course of a Java program, a special class called an `Exception` can be **thrown**, which indicates that something went wrong during the execution of the program. If the exception is not handled anywhere in the program, it will propagate up through the call stack until it is handled by the JVM which then terminates the program.

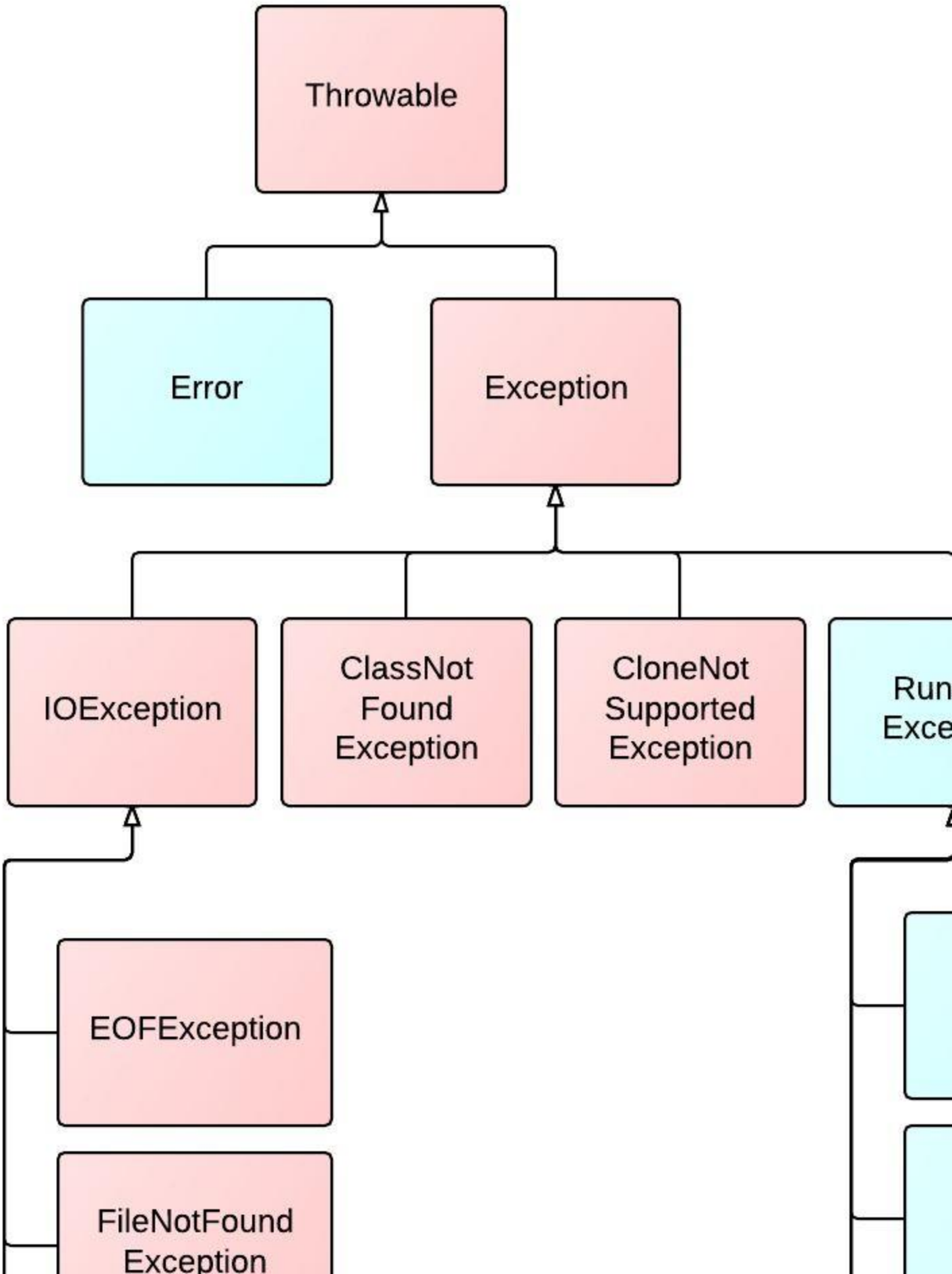
Exceptions Handling / Declaring Exceptions

When risky code is written that has the possibility of throwing an exception, it can be dealt with in one of two ways:

1. Handling means that the risky code is placed inside a try/catch block
2. Declaring means that the type of exception to be thrown is listed in the method signature with the `throws` keyword. This is also called "ducking" the exception - you let the code which calls the method deal with it.

Exception Hierarchy

Exception Class Hierarchy


Throwable
Error
Exception
IOException
ClassNot Found Exception
CloneNot Supported Exception
Run Exce
EOFException
FileNotFound Exception

The exception class hierarchy starts with the `Throwable` class which inherits from `Object`. Any object which is a `Throwable` can be "thrown" in a program by the JVM or by the programmer using the `throws` keyword. The `Exception` and `Error` classes both extend `Throwable`. An `Error` represents something that went so horribly wrong with your application that you should not attempt to recover from. Some examples of errors are:

- `ExceptionInInitializerError`
- `OutOfMemoryError`
- `StackOverflowError`

`Exception` is a general exception class which provides an abstraction for all exceptions. There are many subclasses of `Exception`, as shown above.

Unchecked vs. Checked Exceptions

The `Exception` class and all of its subclasses, except for `RuntimeException`, are known as "checked exceptions". These represent occasions where it is reasonable to anticipate an unexpected condition, like a file not existing when attempting to write to it (which would result in a `FileNotFoundException`). **Checked exceptions are required to be handled or declared by the programmer** - otherwise, the code will not compile.

`RuntimeException` is a special type of exception - it, and all of its subclasses - are known as "unchecked exceptions". An **unchecked exception** is an exception that **is not required to be handled or declared** like checked exceptions are. Some examples include:

- `ArithmeticException` for illegal math operations
- `IndexOutOfBoundsException` for if you reference an index that is greater than the length of an array
- `NullPointerException` for if you attempt to perform an operation on a reference variable that points to a `null` value

Try-with-resources

When using `try/catch` blocks, often times some object used in the code is a resource that should be closed after it is no longer needed to prevent memory leaks - for example a `FileReader`, `InputStream`, or a JDBC `Connection` object. With Java 7, we can use a `try-with-resources` block which will automatically close the resource for us:

Old way

```
try {
  InputStream is = new FileInputStream("./some/file.ext");
  String s = is.read();
} catch(IOException e) {
```

```
} finally {
  is.close();
}
```

New way

```
try(InputStream is = new FileInputStream("./some/file.ext")) {
  String s = is.read();
} catch(IOException e) {}
```

Whatever is placed within the parenthesis of the `try` statement will be closed automatically - thus, we don't need to explicitly call it within our `finally` block above. This new format requires the object in the `try` statement to **implement the AutoCloseable interface**.

Multi-catch blocks

Try/catch/finally Blocks

In order to handle exceptions that could be thrown in our application, a `try/catch` block can be used. The `try` block encloses the code that may throw an exception, and the `catch` block defines an exception to catch and then runs the code inside only if that type of exception is thrown. We can optionally include a `finally` block which will run whether an exception is thrown or not. A simple example is shown below:

```
try {
  object.someRiskyMethodCall();
} catch(Exception e) {
  System.out.println("phew! that was close!");
} finally {
  System.out.println("I'll run whether there was a problem or not!");
}
```

Try/catch/finally & Multi-catch Block Rules

Catch and finally blocks have several different rules which must be followed:

- Multiple catch blocks are allowed. More specific exceptions must come before more general exception types.
- Multi-catch blocks (catching more than one exception in a given block) are allowed, exception types are separated by `||`
- The `finally` block is optional

- A `try/finally` block only IS allowed, but a `try` block by itself is not
- A `finally` block will always execute, unless of course `System.exit()` is called.

Custom Exceptions

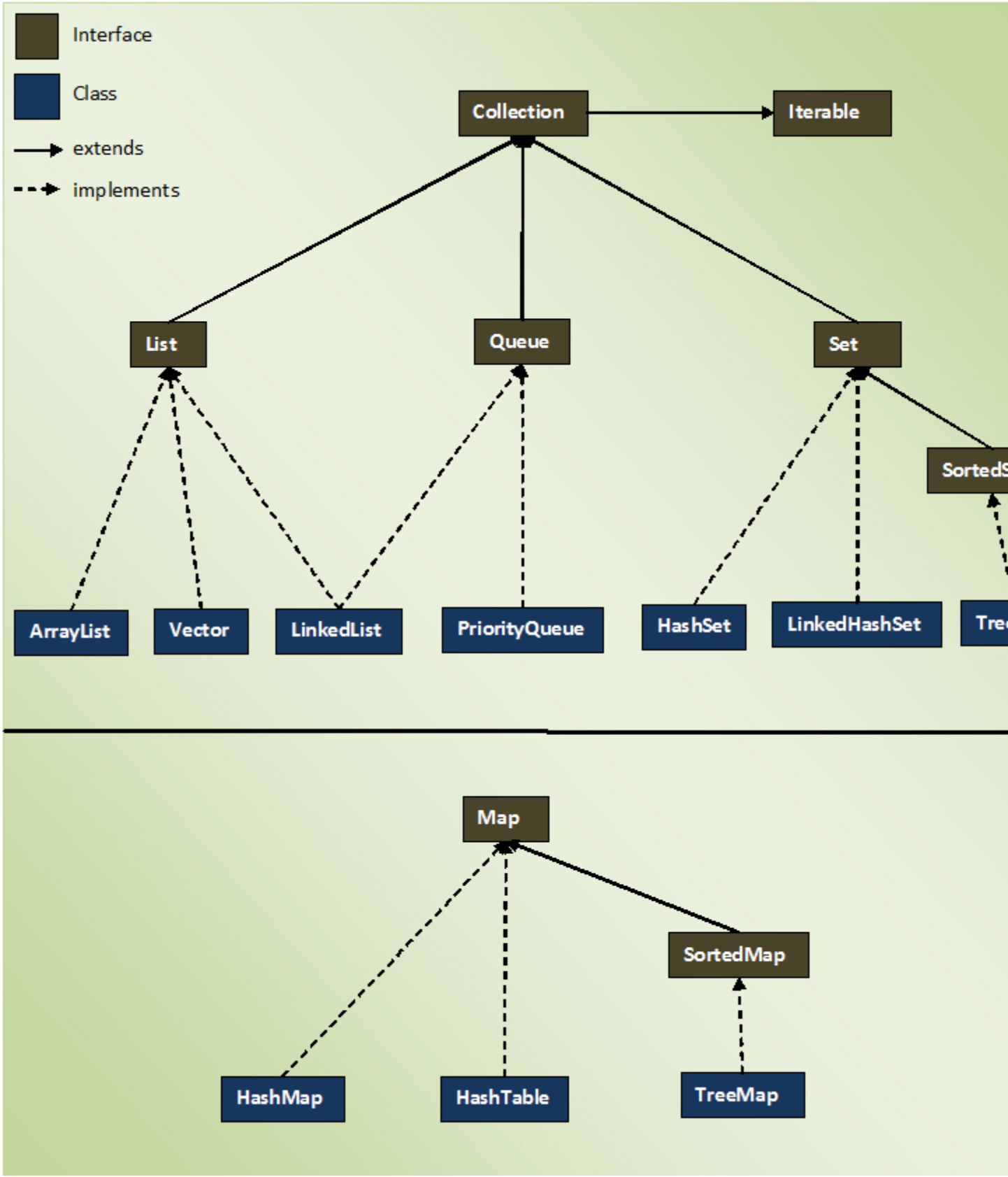
A programmer can create custom exceptions in Java by extending any exception class. If you extend `RuntimeException`, however, you will be creating an unchecked exception. This is a good idea if you do **not** want other code to have to handle your exception being thrown. If you do always want to require your exception to be handled, then create a checked exception by extending any existing one, or the `Exception` class itself.

```
public class MyCheckedException extends Exception {}
public class MyUncheckedException extends RuntimeException {}


public class ExceptionThrower {


  public static void main(String[] args) {
    try {
          throw new MyCheckedException("uh oh");
        } catch(MyCheckedException e) {} // we're just ignoring it here


    if ( 100 > 1) {
          throw new MyUncheckedException("you're not required to handle me!");
        }
  }


  public static void declareChecked() throws MyCheckedException {
    throw new MyCheckedException("this one is declared!");
  }
}
```

Collection API

Collections Framework

The Collections framework in Java is a set of classes and interfaces that implement commonly used data structures. A collection is a single object which acts as a container for other objects. The Collections API is organized in a class hierarchy shown in simplified version below:


Interface
Class
extends
implements
Collection
Iterable
List
Queue
Set
ArrayList
Vector
LinkedList
PriorityQueue
HashSet
LinkedHashSet
Map
SortedMap
HashMap
HashTable
TreeMap

The important interfaces in the Collections API are:

- `Iterable` - guarantees the collection can be iterated over
- `List` - an ordered collection
- `Set` - a collection does not contain duplicates
- `Queue` - a collection that operates on a first-in-first-out (FIFO) basis
- `Map` - contains key/value pairs. Does not extend `Iterable`.

Collections Class

The Collections class - not to be confused with the Collection interface - defines many `static` helper methods which operate on any given collection. Use this class for help with sorting, searching, reversing, or performing other operations on collections.

Lists

List Interface

A `List` is a collection that is ordered and preserves the order in which elements are inserted into the list. Contrary to sets, duplicate entries are allowed. Also, elements are accessed by their index, which begins with 0 for the first element in the list.

Vector

`Vector` is an older class which implements `List` - it is essentially a thread-safe implementation of an `ArrayList`.

Stack

`Stack` is an older implementation of the stack data structure (last-in-first-out, or LIFO). Generally you should use an `ArrayDeque` for implementing a stack.

Sets

Set Interface

`Set` is an interface which defines a data structure which:

- is NOT index driven
- only allows unique elements
- generally DOES NOT preserve the order in which elements were inserted

Queues

Queue Interface

A `Queue` is a data structure used when elements should be added and removed in a specific order. Unless specified, elements are ordered FIFO (first-in-first-out). Some useful methods declared are:

- `offer()`
- `peek()`
- `poll()`

Maps

Map Interface

Map does not implement the Collection interface, however it is considered to be part of the Collections framework. It is used to identify a value by a key, and each entry in a map is a key-value pair. Because it does not implement `Iterable`, Maps cannot be iterated over directly. Instead, one must either:

- use the `entrySet()` method to iterate over the set of `Map.Entry`
- use the `keySet()` method to iterate over the set of keys
- use the `values()` method to return a `Collection` of values which can then be iterated over

HashMap

`HashMap` is a Map which:

- Stores elements in key-value pairs
- Insertion/Retrieval of element by key is fast
- Tradeoff is that it does not maintain the order of insertion
- Permits one null key and null values

TreeMap

`TreeMap` is a Map whose:

- Keys are stored in a Sorted Tree structure
- Main benefit is that keys are always in a sorted order
- Insertion/Retrieval are slow
- Cannot contain null keys as null cannot be compared for sorting

HashTable

`HashTable` is an older, thread-safe implementation of a `HashMap`. It does not allow null keys or null values.

HashSet and TreeSet

HashSet

A `HashSet` implements `Set` and is backed by a `HashMap`. It:

- Guarantees no ordering when iterating
- Allows one `null` value
- Allows fast insertion and traversal
- Does not maintain order in which you insert elements

TreeSet

A `TreeSet` is a `Set` whose elements maintain sorted order when inserted. Internally, it is backed by a Sorted Tree. Insertion and removal of elements is slow, because the elements must maintain sorted order. It cannot contain any `null` values, since `null` cannot be compared to any object.

ArrayList and LinkedList

ArrayList

An `ArrayList` is a concrete class which implements `List`. It is a data structure which contains an array within it, but can resize dynamically. Once it reaches maximum capacity, an `ArrayList` will increase its size by 50% by copying its elements to a new (internal) array. Traversal is fast (constant time) because elements can be randomly accessed via index, just like an array. Insertion or removal of elements is slow, however (linear time, since there is a risk of having to resize the underlying array).

LinkedList

A `LinkedList` implements both the `List` and `Queue` interfaces, so it has all methods in both interfaces. The data structure is composed of nodes internally, each with a reference to the previous node and the next node - i.e. a doubly-linked list. Because of this structure, insertion or removal of elements is fast (no risk to resize, just change the nearest references), but traversal is slow for an arbitrary index.

ArrayDeque and PriorityQueue

ArrayDeque

Pronounced as 'array-deck', this concrete class can be implemented for either the queue or stack data structure. It is an implementation of a pure double-ended queue (elements can be added or removed from either end of the queue). An `ArrayDeque` stores elements in a resizable array internally, and it has a few extra useful methods defined:

- `pop()`
- `push()`
- `peekFirst()`
- `peekLast()`
- `pollFirst()`
- `pollLast()`
- `offerFirst()`
- `offerLast()`

Operation	Throws Exception	Returns null
Insert	`boolean add(E e)`	`boolean offer(E e)`
Remove	`E remove()`	`E poll()`
Examine	`E element()`	`E peek()`

PriorityQueue

We know that `Queue` serves the requests based on FIFO(First in First out) but sometimes the elements of the queue are needed to be processed according to the priority, that's when the `PriorityQueue` comes into a picture.

A `PriorityQueue` serves the requests based on the priority heap. The elements of the priority queue are ordered according to their natural ordering, or by a `Comparator` provided at queue construction time, depending on which constructor is used. A priority queue does not permit `null` elements. A priority queue relying on natural ordering also does not permit insertion of non-comparable objects (doing so may result in `ClassCastException`).

Methods of PriorityQueue Class

- `boolean add(E e)` - Inserts the specified element into this priority queue.
- `void clear()` - Removes all of the elements from this priority queue.
- `Comparator<? super E> comparator()` - Returns the comparator used to order the elements in this queue, or null if this queue is sorted according to the natural ordering of its elements.
- `boolean contains(Object o)` -Returns true if this queue contains the specified element.
- `Iterator<E> iterator()` -Returns an iterator over the elements in this queue.
- `boolean offer(E e)` -Inserts the specified element into this priority queue.

- `E peek()` -Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.
- `E poll()` -Retrieves and removes the head of this queue, or returns null if this queue is empty.
- `boolean remove(Object o)` -Removes a single instance of the specified element from this queue, if it is present.
- `int size()` -Returns the number of elements in this collection.
- `Object[] toArray()` -Returns an array containing all of the elements in this queue.
- `<T> T[] toArray(T[] a)` -Returns an array containing all of the elements in this queue; the runtime type of the returned array is that of the specified array.

Comparable Interface

`Comparable` is an interface which defines the natural ordering for a class. A class must implement `Comparable` if it is to be sorted by the `compareTo()` method.

```
public interface Comparable<T> {

        public int compareTo(T o);

}
```

The `compareTo()` method returns an `int` which is:

- Zero, if the two objects are equal
- Negative, if this object is smaller than that
- Positive, if this object is greater than that

Comparator Interface

`Comparator` is an interface which allows you to define a total ordering on some collection of objects. A class that is to be sorted by `compare()` does not have to implement `Comparator`.

```
public interface Comparator<T> {

        public int compare(T firstObject, T secondObject);

}
```

Generics

Generics are constructs introduced in Java 5 which enforce compile time safety by allowing you to use parameterized types. They are covered here because they are frequently and

heavily used with collections. Generics can be declared on a class (generic types), method parameters (generic methods), or return types.

Before Java 5, you had to write something like this and hope other developers understood to only put Strings inside:

```
List names = new ArrayList();

names.add("Alice"); // good use

name.add(new Object()); // uh oh - we want to prevent this from happening
```

With generics, you can restrict a class to only accept objects of a given type and the compiler will prevent you from using any other type:

```
List<String> names = new ArrayList<>(); // using a List of Strings only

names.add("Alice"); // nice!

names.add(new Object()); // now we get a compilation error to stop this - generics sa
ve the day!
```

Generic Classes

To make a class (or interface) generic, use the angle brackets when declaring it, and use an arbitrary "generic type" which is determined by the invoking code. The generic type can then be reused throughout the class to enforce type safety.

```
public class MyGenericClass<T> {
  private T instance;


  // simple generic setter method
  public void setObject(T object) {
    this.instance = object;
  }
}
```

Naming Convention for Generics

Technically, type parameters can be named anything you want. The convention is to use single, uppercase letters to make it obvious that they are not real class names.

- E => Element
- K => Map Key
- V => Map Value
- N => Number
- T => Generic data type
- S, U, V, and so on => For multiple generic data types