Collection Framework Generics & Annotations

Collection of Objects

- In Java, you can collect objects using various collection classes provided by the Java Collections Framework, such as ArrayList, LinkedList, HashSet, HashMap, and more.
- These collection classes allow you to store, manipulate, and retrieve groups of objects.
- Collections can store objects of different types, including custom objects, and provide methods for adding, removing, and iterating through the elements.
- Collections can be used to represent lists, sets, maps, and other data structures, making them essential for managing and organizing data in Java applications.

Collection Interfaces & Hierarchy

- Java provides a hierarchy of collection interfaces that define common behaviors and contracts for different types of collections.
- At the top of the hierarchy is the Collection interface, which represents a basic collection of objects. It is extended by more specialized interfaces.
  - List, Set, and Queue are three primary subinterfaces of Collection.
  - List represents an ordered collection with duplicate elements.
  - Set represents an unordered collection with unique elements.
  - Queue represents a collection designed for holding elements before processing them.
- Map is a separate interface for key-value pair mappings and is not part of the Collection hierarchy.
- Other specialized interfaces like SortedSet, NavigableSet, and Deque further extend these core interfaces to provide additional functionality.
- These interfaces define common methods for adding, removing, querying, and iterating through collections, ensuring consistency and ease of use when working with different collection types.

List, Set & Map

List, Set, and Map are three fundamental collection types in Java, each serving different purposes:

- **List**: Represents an ordered collection of elements where duplicates are allowed. Lists are typically used when the order of elements matters, and you need to access elements by index. Common implementations include ArrayList and LinkedList.
- **Set**: Represents an unordered collection of unique elements. Sets are used when you need to ensure that each element is unique within the collection. Common implementations include HashSet, LinkedHashSet, and TreeSet.
- **Map**: Represents a collection of key-value pairs, where each key is associated with a value. Maps are used when you need to perform lookups based on keys. Common implementations include HashMap, LinkedHashMap, and TreeMap.

Types of List

ArrayList:	• An ArrayList is a dynamic array-based implementation of the List interface. • It provides fast random access to elements and is efficient for adding or removing elements at the end. • Elements can be added and removed anywhere in the list.
LinkedList:	• A LinkedList is implemented as a doubly-linked list. • It provides efficient element insertion and removal at both ends of the list. • It is useful when frequent insertions and removals are required.
Vector:	• A Vector is similar to an ArrayList but is synchronized, making it thread-safe. • It is less commonly used in modern Java development due to the overhead of synchronization.
Stack:	• A Stack is a specialized list that follows the Last-In-First-Out (LIFO) order. • It is often used for implementing algorithms like depth-first search (DFS) and managing function calls in recursion.
CopyOnWriteArrayList:	• A CopyOnWriteArrayList is thread-safe and allows concurrent read operations without locking. • Write operations (add, set, remove) create a new copy of the list, making them slower.
Immutable List (e.g., Collections.unmodifiableList):	• An immutable list cannot be modified after it is created. • Changes to the list result in a new list being created, which makes it safe for concurrent access.

Types of Set

Hashset:	• A HashSet is an unordered collection of unique elements. • It provides constant-time performance for basic operations like add, remove, and contains. • Elements are stored in a hash table, making it efficient for lookups but not maintaining insertion order.
LinkedHashSet:	• A LinkedHashSet is similar to a HashSet, but it maintains the order of elements as they were added. • It also ensures uniqueness, like HashSet.
TreeSet:	• A TreeSet is an ordered set that uses a Red-Black tree structure to store elements in sorted order. • Elements are automatically sorted upon insertion. • It is useful when you need elements in a specific order.
EnumSet:	• An EnumSet is a specialized set designed to work with enum types. • It is highly efficient and compact for enum types, offering constant-time performance for most operations.
CopyOnWriteArraySet:	• A CopyOnWriteArraySet is thread-safe and allows concurrent read operations without locking. • Write operations (add, remove) create a new copy of the set, making them slower.
Immutable Set (e.g., Collections.unmodifiableSet):	• An immutable set cannot be modified after it is created. • Changes to the set result in a new set being created, which makes it safe for concurrent access.

Types of Map

HashMap:

- A HashMap is an unordered collection of key-value pairs.
- It provides constant-time performance for basic operations like put, get, and remove.
- Elements are stored in a hash table, making it efficient for lookups but not maintaining insertion order or sorted order.

LinkedHashMap:

- A LinkedHashMap is like a HashMap, but it maintains the order of key-value pairs as they were inserted.
- It combines the fast lookups of a HashMap with predictable iteration order.

TreeMap:

- A TreeMap is an ordered map that uses a Red-Black tree structure to store key-value pairs in sorted order.
- Keys are automatically sorted upon insertion, making it useful when you need key-value pairs in a specific order.

EnumMap:

- An EnumMap is a specialized map designed to work with enum types as keys.
- It is highly efficient and compact for enum types, offering constant-time performance for most operations.

WeakHashMap:

- A WeakHashMap is a map in which the keys are weakly referenced. This means that if a key is no longer strongly referenced, it may be automatically removed from the map.
- It is often used for caching or managing resources that need to be automatically cleaned up when they are no longer in use.

IdentityHashMap:

- An IdentityHashMap uses reference equality (==) rather than object equality (equals) to compare keys.
- It is useful when you want to maintain separate entries for objects that are equal according to their equals method but have different identity.

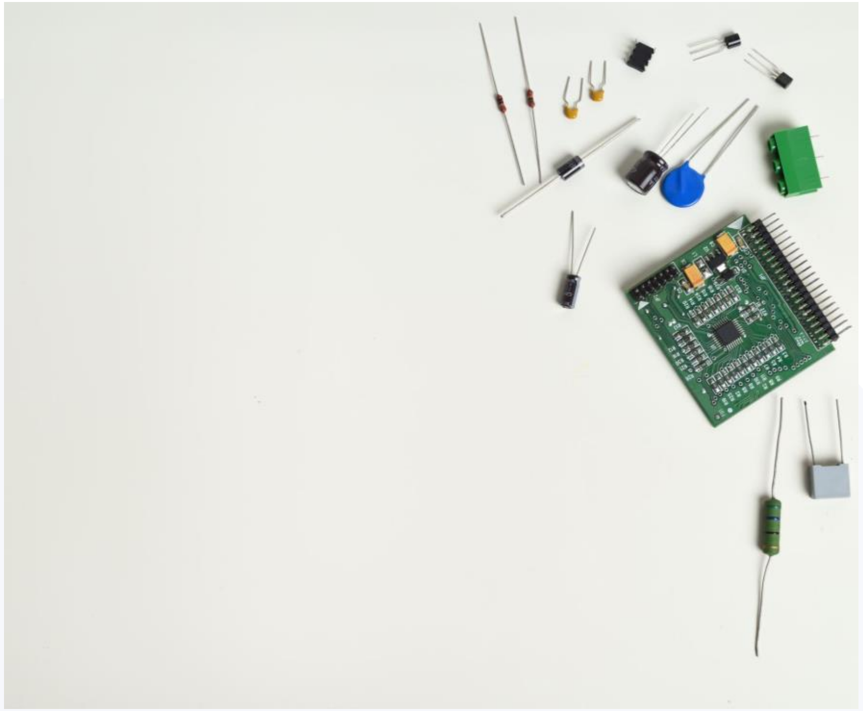
Hashtable:

- A Hashtable is like a HashMap but is synchronized, making it thread-safe.
- It is less commonly used in modern Java development due to the overhead of synchronization.

Iterator

An iterator in Java is an interface provided by the Java Collections Framework that allows you to traverse and manipulate the elements of a collection (such as a list, set, or map) sequentially. It provides a way to access elements one at a time without exposing the underlying data structure. The Iterator interface includes the following methods:

- **Boolean hasNext()**: Checks if there are more elements in the collection. Returns true if there are more elements, otherwise false.
- **E next()**: Returns the next element in the collection and advances the iterator. If there are no more elements, it throws a NoSuchElementException.
- **Void remove()**: Removes the last element returned by next() from the underlying collection (optional operation). It can throw an Unsupported Operation Exception if the collection does not support removal.



Generics

Generics in Java allow you to create classes, interfaces, and methods that operate on objects of various types while providing compile-time type safety. They enable you to write code that is more flexible, reusable, and less error-prone. Generics are commonly used in Java collections and other library classes to work with different types of data. Here are some key points about generics:

- **Type Safety**: Generics provide type safety by allowing you to specify the data types of objects that can be used with a class, method, or interface. This ensures that the code is checked at compile time, reducing the chances of runtime type-related errors.
- **Parameterized Types**: Generics use parameterized types, which are specified within angle brackets (<>). For example, List<String> is a parameterized type where String is the type parameter.
- **Class Generics**: You can create generic classes by using type parameters. For example, you can create a generic Box<T> class to hold objects of any type T.
- **Method Generics**: You can also define generic methods within non-generic classes. These methods can have their own type parameters that are independent of the class's type parameters.
- **Wildcard Types**: Wildcard types (e.g., ?) allow you to work with unknown types in a generic class or method. They provide flexibility when the exact type is not known or important.
- **Bounded Type Parameters**: You can restrict the types that can be used as type parameters by using bounded type parameters. For example, <? extends Number> restricts the parameter to types that are subclasses of Number.
- **Generic Interfaces**: You can create generic interfaces, such as Comparable<T>, which allows objects of different types to be compared.
- **Type Erasure**: Java uses type erasure during compilation to replace generic types with their raw types (e.g., List instead of List<String>). This ensures backward compatibility with older versions of Java.

Annotations

Annotations in Java are metadata that provide information about the code to the compiler, runtime, or other tools. They allow you to add additional information or behavior to your code without affecting the core functionality. Annotations are represented by @ symbols followed by an annotation name, and they can be applied to classes, methods, fields, and other program elements. Here are some key points about annotations in Java:

- **Built-in Annotations**: Java provides several built-in annotations that serve various purposes, such as @Override, @Deprecated, and @SuppressWarnings. These annotations help the compiler and developers understand code better.
- **Custom Annotations**: You can create custom annotations by defining your annotation type. Custom annotations are often used for code documentation, code generation, or to control the behavior of custom tools.
- **Retention Policy**: Annotations can have different retention policies:
  - SOURCE: Annotations are only available in the source code and are not included in the compiled class files.
  - CLASS: Annotations are included in the class files but are not accessible at runtime.
  - RUNTIME: Annotations are available at runtime through reflection, allowing you to inspect and modify them.
- **Target Elements**: Annotations can be applied to various program elements, including classes, methods, fields, parameters, packages, and more. The set of elements to which an annotation can be applied is defined by its @Target meta-annotation.
- **Meta-Annotations**: Meta-annotations are annotations that annotate other annotations. They provide additional information about how annotations should be processed. Examples include @Target, @Retention, and @Documented.
- **Annotation Processors**: Annotation processors are tools that process annotations and generate code or perform other tasks based on the annotations present in the code. They are commonly used for code generation and code analysis.
- **Examples of Use**: Annotations are used in various contexts, such as:
  - Marking methods as overridden with @Override.
  - Indicating that a method is deprecated with @Deprecated.
  - Suppressing warnings with @SuppressWarnings.
  - Configuring dependency injection in frameworks like Spring with custom annotations.

xebia.com