

MyJavaNotes

Constructors

- They have the same name as the class.
- They are used to create an instance of this class in main
- A default constructor with no parameters exists **unless you make your own custom constructor with desired number and types of parameters**

```
• public defaultConstructor() {  
• }
```

- If you don't a class to not have instances, you can just create a custom default constructor but private one.... This way it wont be accessed from main

```
• private defaultConstructor() {  
• }
```

.....

Abstract Classes and Methods

<https://www.youtube.com/watch?v=HvPIEJ3LHgE&t=26s>

Abstract Class **has to do with implementation**

- You can't instantiate this class
- It contains abstract methods, which are not implemented, they are implemented in the subclass
- May include both implemented methods and unimplemented methods
- Can have fields, constructors and methods that are either abstract or non-abstract
- Useful for defining a base class

Abstract Method

- This is a method with no body/implementation
- Subclasses are forced to implement these kind of methods
- Must be implemented in non abstract subclass
- Only appears within abstract classes or in interfaces

Static vs Non-Static Variables and Methods

<https://www.youtube.com/watch?v=-Y67pdWHr9Y&t=10s>

Static Variable

- Belong to the class itself.. no need to create an instance of the class to access this kind of variables
- It is shared across all the instances/copies of the class. To change it across all the kids/copies, you must change it in the main class
- Saves memory coz it is only declared once
- Declared with static keyword. like `static int count;`

Static Method

has to do with belonging to the class

- Belong to the class rather than instances
- You don't have to create an instance of the class to access this method.
- Commonly used as utility methods e.g. `Math.pow ()`

```
static void print() {  
}
```

Non-Static Variable

- Also known as instance variables
- They are unique to each object

Non-Static Method

- Also known as instance methods
- Can access both non-static (instance) variables and methods, as well as static variables and methods.

Customs Exceptions

NNNN

- Also known as

Polymorphism

<https://www.youtube.com/watch?v=jhDUxynEQRI&t=2s>

NNNN

- Also known as

Generics

Extra


- They offer code reusability
- They offer type safety
- Work with wrapper types not primitives
- Parameters can be more than 1 or even be bounded
- **Commonly used with** Java collections (e.g., `List<T>`, `Map<K, V>`, `Set<T>`)
- **Elimination of Type Casting.** no need for explicit casting coz you know the type you are dealing with.

In Java, a non-generic list can hold any type of object, but when retrieving elements, you might not know their types. This uncertainty can lead to errors and requires manual typecasting for each object, which is tedious and error-prone.

Generics solve this by letting you specify a list's type up front. This ensures type safety, removes the need for casting, and makes code cleaner and safer

Example Without Generics:

java


 Copy code

```
List list = new ArrayList();
list.add("Hello");
list.add(10); // Adds an Integer

// Casting is needed to avoid errors:
String str = (String) list.get(0);
Integer num = (Integer) list.get(1);
```

Example With Generics:

java

 Copy code

```
List<String> list = new ArrayList<>();  
list.add("Hello");  
// list.add(10); // Compile-time error: Only Strings are allowed  
  
String str = list.get(0); // No casting needed, type is known
```

Type Parameters < >

- Common conventions:

T – Type **E – Element (for collections)** **K – Key** **V – Value**

- Here you know the datatype that's going to replace the T
- If you don't know then,,,,, **wildcard**

Generic Classes

- Uses is generics to work with a range of datatypes.
- T is a generic type acting as any datatype that will be passed when an instance of this generic class is created in main or wherever.
- In main, you just create the instance of this class with the datatype that you want the class to work with.

```
public class calc<T> {
    T num;

    public calc(T num) {
        this.num = num;
    }
}
```

- In main

```
calc<Integer> c = new calc<>(2025); for integers
```

```
calc<Double> c = new calc<>(2,95); for doubles
```

Generic Method

- They offer code reusability

```
public static <T> void printArray(T[] array) {
    for (T item : array) {
        System.out.println(item);
    }
}
```

Bounded Types

- Here the generic is restricted to a specific type
- type parameter T **can only be replaced** by a class that is either Number itself or a subclass of Number

```
public class calc<T extends number> {
}
```

Wildcard ?

- Here you don't know the datatype that's going to replace the T.. so use wildcard instead of type parameter

```
public void printNumbers(List<? extends Number> list) {  
    • for (Number n : list) {  
        System.out.println(n);  
    }  
}
```

Here remember that List is a generic interface..

Bounded wildcards

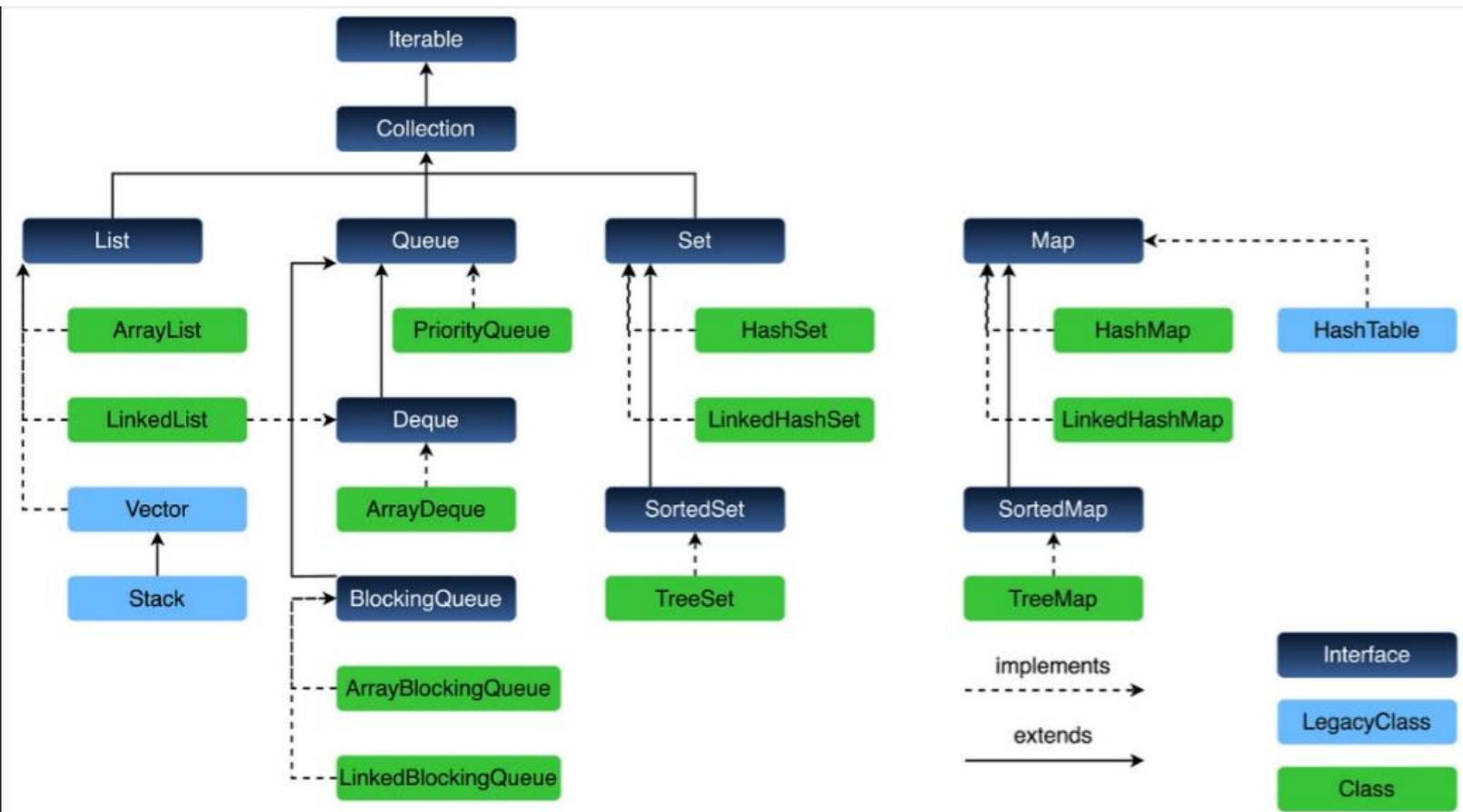
? extends T

- Accepts T or its subclasses
- Useful for **read-only** operations

? super T

- Accepts T or its superclasses
- Useful for **write-only** operations

Java Collection



List<E> Interface methods

All Methods	Instance Methods	Abstract Methods	Default Methods
Modifier and Type		Method and Description	
boolean		add(E e) Appends the specified element to the end of this list (optional operation).	
void		add(int index, E element) Inserts the specified element at the specified position in this list (optional operation).	
boolean		addAll(Collection<? extends E> c) Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator (optional operation).	
boolean		addAll(int index, Collection<? extends E> c) Inserts all of the elements in the specified collection into this list at the specified position (optional operation).	
void		clear() Removes all of the elements from this list (optional operation).	
boolean		contains(Object o) Returns true if this list contains the specified element.	
boolean		containsAll(Collection<?> c) Returns true if this list contains all of the elements of the specified collection.	
boolean		equals(Object o) Compares the specified object with this list for equality.	
E		get(int index) Returns the element at the specified position in this list.	

int	hashCode() Returns the hash code value for this list.
int	indexOf(Object o) Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
boolean	isEmpty() Returns true if this list contains no elements.
Iterator<E>	iterator() Returns an iterator over the elements in this list in proper sequence.
int	lastIndexOf(Object o) Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
ListIterator<E>	listIterator() Returns a list iterator over the elements in this list (in proper sequence).
ListIterator<E>	listIterator(int index) Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list.
E	remove(int index) Removes the element at the specified position in this list (optional operation).
boolean	remove(Object o) Removes the first occurrence of the specified element from this list, if it is present (optional operation).

boolean	removeAll(Collection<?> c) Removes from this list all of its elements that are contained in the specified collection (optional operation).
default void	replaceAll(UnaryOperator<E> operator) Replaces each element of this list with the result of applying the operator to that element.
boolean	retainAll(Collection<?> c) Retains only the elements in this list that are contained in the specified collection (optional operation).
E	set(int index, E element) Replaces the element at the specified position in this list with the specified element (optional operation).
int	size() Returns the number of elements in this list.
default void	sort(Comparator<? super E> c) Sorts this list according to the order induced by the specified Comparator .
default Spliterator<E>	spliterator() Creates a Spliterator over the elements in this list.
List<E>	subList(int fromIndex, int toIndex) Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.
Object[]	toArray() Returns an array containing all of the elements in this list in proper sequence (from first to last element).
<T> T[]	toArray(T[] a) Returns an array containing all of the elements in this list in proper sequence (from first to last element); the runtime type of the returned array is that of the specified array.

Map<K,V> Interface Methods

All Methods	Instance Methods	Abstract Methods	Default Methods
Modifier and Type	Method and Description		
void	clear() Removes all of the mappings from this map (optional operation).		
default V	compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction) Attempts to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping).		
default V	computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction) If the specified key is not already associated with a value (or is mapped to null), attempts to compute its value using the given mapping function and enters it into this map unless null.		
default V	computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction) If the value for the specified key is present and non-null, attempts to compute a new mapping given the key and its current mapped value.		
boolean	containsKey(Object key) Returns true if this map contains a mapping for the specified key.		
boolean	containsValue(Object value) Returns true if this map maps one or more keys to the specified value.		
Set<Map.Entry<K,V>>	entrySet() Returns a Set view of the mappings contained in this map.		
boolean	equals(Object o) Compares the specified object with this map for equality.		

default void	forEach (BiConsumer <? super K ,? super V > action) Performs the given action for each entry in this map until all entries have been processed or the action throws an exception.
V	get (Object key) Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
default V	getOrDefault (Object key, V defaultValue) Returns the value to which the specified key is mapped, or defaultValue if this map contains no mapping for the key.
int	hashCode () Returns the hash code value for this map.
boolean	isEmpty () Returns true if this map contains no key-value mappings.
Set < K >	keySet () Returns a Set view of the keys contained in this map.
default V	merge (K key, V value, BiFunction <? super V ,? super V ,? extends V > remappingFunction) If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.
V	put (K key, V value) Associates the specified value with the specified key in this map (optional operation).
void	putAll (Map <? extends K ,? extends V > m) Copies all of the mappings from the specified map to this map (optional operation).

default V	putIfAbsent (K key, V value) If the specified key is not already associated with a value (or is mapped to null) associates it with the given value and returns null, else returns the current value.
V	remove (Object key) Removes the mapping for a key from this map if it is present (optional operation).
default boolean	remove (Object key, Object value) Removes the entry for the specified key only if it is currently mapped to the specified value.
default V	replace (K key, V value) Replaces the entry for the specified key only if it is currently mapped to some value.
default boolean	replace (K key, V oldValue, V newValue) Replaces the entry for the specified key only if currently mapped to the specified value.
default void	replaceAll (BiFunction <? super K ,? super V ,? extends V > function) Replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.
int	size () Returns the number of key-value mappings in this map.
Collection < V >	values () Returns a Collection view of the values contained in this map.

Queue<E> Interface Methods

Method Summary

All Methods	Instance Methods	Abstract Methods
Modifier and Type	Method and Description	
boolean	add(E e)	Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning true upon success and throwing an <code>IllegalStateException</code> if no space is currently available.
E	element()	Retrieves, but does not remove, the head of this queue.
boolean	offer(E e)	Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions.
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.
E	remove()	Retrieves and removes the head of this queue.

Deque<E> Interface Methods

All Methods	Instance Methods	Abstract Methods
Modifier and Type	Method and Description	
boolean	add(E e)	Inserts the specified element into the queue represented by this deque (in other words, at the tail of this deque) if it is possible to do so immediately without violating capacity restrictions, returning true upon success and throwing an <code>IllegalStateException</code> if no space is currently available.
void	addFirst(E e)	Inserts the specified element at the front of this deque if it is possible to do so immediately without violating capacity restrictions, throwing an <code>IllegalStateException</code> if no space is currently available.
void	addLast(E e)	Inserts the specified element at the end of this deque if it is possible to do so immediately without violating capacity restrictions, throwing an <code>IllegalStateException</code> if no space is currently available.
boolean	contains(Object o)	Returns true if this deque contains the specified element.
Iterator<E>	descendingIterator()	Returns an iterator over the elements in this deque in reverse sequential order.
E	element()	Retrieves, but does not remove, the head of the queue represented by this deque (in other words, the first element of this deque).
E	getFirst()	Retrieves, but does not remove, the first element of this deque.
E	getLast()	Retrieves, but does not remove, the last element of this deque.

Iterator<E>	iterator() Returns an iterator over the elements in this deque in proper sequence.
boolean	offer(E e) Inserts the specified element into the queue represented by this deque (in other words, at the tail of this deque) if it is possible to do so immediately without violating capacity restrictions, returning true upon success and false if no space is currently available.
boolean	offerFirst(E e) Inserts the specified element at the front of this deque unless it would violate capacity restrictions.
boolean	offerLast(E e) Inserts the specified element at the end of this deque unless it would violate capacity restrictions.
E	peek() Retrieves, but does not remove, the head of the queue represented by this deque (in other words, the first element of this deque), or returns null if this deque is empty.
E	peekFirst() Retrieves, but does not remove, the first element of this deque, or returns null if this deque is empty.
E	peekLast() Retrieves, but does not remove, the last element of this deque, or returns null if this deque is empty.
E	poll() Retrieves and removes the head of the queue represented by this deque (in other words, the first element of this deque), or returns null if this deque is empty.
E	pollFirst() Retrieves and removes the first element of this deque, or returns null if this deque is empty.

E	pollLast() Retrieves and removes the last element of this deque, or returns null if this deque is empty.
E	pop() Pops an element from the stack represented by this deque.
void	push(E e) Pushes an element onto the stack represented by this deque (in other words, at the head of this deque) if it is possible to do so immediately without violating capacity restrictions, throwing an <code>IllegalStateException</code> if no space is currently available.
E	remove() Retrieves and removes the head of the queue represented by this deque (in other words, the first element of this deque).
boolean	remove(Object o) Removes the first occurrence of the specified element from this deque.
E	removeFirst() Retrieves and removes the first element of this deque.
boolean	removeFirstOccurrence(Object o) Removes the first occurrence of the specified element from this deque.
E	removeLast() Retrieves and removes the last element of this deque.
boolean	removeLastOccurrence(Object o) Removes the last occurrence of the specified element from this deque.
int	size() Returns the number of elements in this deque.

Set<E> Interface Methods

All Methods	Instance Methods	Abstract Methods	Default Methods
Modifier and Type		Method and Description	
boolean		add(E e) Adds the specified element to this set if it is not already present (optional operation).	
boolean		addAll(Collection<? extends E> c) Adds all of the elements in the specified collection to this set if they're not already present (optional operation).	
void		clear() Removes all of the elements from this set (optional operation).	
boolean		contains(Object o) Returns true if this set contains the specified element.	
boolean		containsAll(Collection<?> c) Returns true if this set contains all of the elements of the specified collection.	
boolean		equals(Object o) Compares the specified object with this set for equality.	
int		hashCode() Returns the hash code value for this set.	
boolean		isEmpty() Returns true if this set contains no elements.	

Iterator<E>	iterator() Returns an iterator over the elements in this set.
boolean	remove(Object o) Removes the specified element from this set if it is present (optional operation).
boolean	removeAll(Collection<?> c) Removes from this set all of its elements that are contained in the specified collection (optional operation).
boolean	retainAll(Collection<?> c) Retains only the elements in this set that are contained in the specified collection (optional operation).
int	size() Returns the number of elements in this set (its cardinality).
default Spliterator<E>	spliterator() Creates a <code>Spliterator</code> over the elements in this set.
Object[]	toArray() Returns an array containing all of the elements in this set.
<T> T[]	toArray(T[] a) Returns an array containing all of the elements in this set; the runtime type of the returned array is that of the specified array.

Array vs ArrayList

<https://www.youtube.com/watch?v=NbYgm0r7u6o&t=40s>

Type Parameterst

2nd

3rd

Wildcard

2nd

3rd

Set and HashSet

<https://www.youtube.com/watch?v=QvHBHuuddYk>

Type Parameterst

2nd

3rd

Wildcard

2nd

3rd

Map and HashMap

<https://www.youtube.com/watch?v=H62Jfv1DJlU&t=26s>

Type Parameterst

2nd

3rd

Wildcard

2nd

3rd