

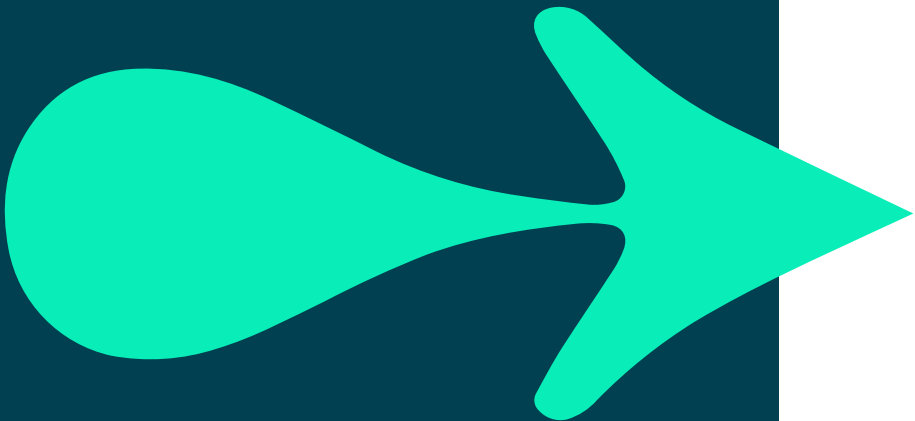


Collections & Generics



CONTENTS

- **Objectives**
 - Compare functionality offered by arrays & collections
 - Understand generic concepts, use generic types & syntax
- **Contents**
 - Recap arrays, introduce collection classes
 - Generic concepts
 - Collections Framework Generic classes
- **Two hands on Labs**



Arrays vs Collection classes

- **Limitations of Arrays**

- Fixed size, (until resized), can't append, insert or delete
- No built-in method to reject duplicates
- Must continually watch out for [ArrayIndexOutOfBoundsException](#)
- But are type-safe!, a Car array can only contain Car references

- **Java offers a collection classes**



- Queue, Stack, List, Set, Map, Dictionary, SortedList ...
 - Know their Capacity & Count
 - Support append, insertions / deletions / searching
- Generic version of these are type-safe

Collection classes in Java 5

Developers had to use the `ArrayList` which could only hold a collection of `Object` type

```
ArrayList myList = new ArrayList();  
myList.add(123);  
myList.add("Bob");  
myList.add(new Car());
```

Java's `get()` method returns an `Object` at an index

<code>int id = myList.get(0);</code>		needs casting. <code>get()</code> returns an <code>Object</code>
<code>int id = (int)myList.get(0);</code>		Cast is valid. It takes time to cast <code>Object</code>
<code>int id = (int)myList.get(1);</code>		Compiles but crashes during runtime

Generic collection classes

- Can hold a collection of a specific type and always returns the expected type

```
ArrayList<Integer> numbers = new ArrayList<>();
```

```
numbers.add(123); ✓ can add an integer
```

```
numbers.add(123.5); ✗ double is not allowed
```

```
numbers.add("Bob"); ✗ Only integers
```

```
numbers.add(new Car()); ✗ no cars!
```

```
int k = numbers.get(0); ✓ no casting is required
```

```
import java.util.ArrayList
```

```
ArrayList<Person> people = new ArrayList<>();
```

```
people.add(new Person("Bob")); ✓ can add a Person type
```

```
Person person = new Person("Linda");
```

```
people.add(person); ✓
```

```
Person p = people.get(0); ✓ no casting is required
```

Iterating through an generic ArrayList

```
ArrayList<String> friends = new ArrayList<>();  
friends.add("Tom");  
friends.add("Sue");  
friends.add("Sanjeev");
```

add() expects a String

```
for (String name : friends)  
    System.out.printf(name);
```

enumerable

```
for(int i = 0; i < friends.size(); i++)  
    System.out.printf( friends.get(i) );
```

Note the size()

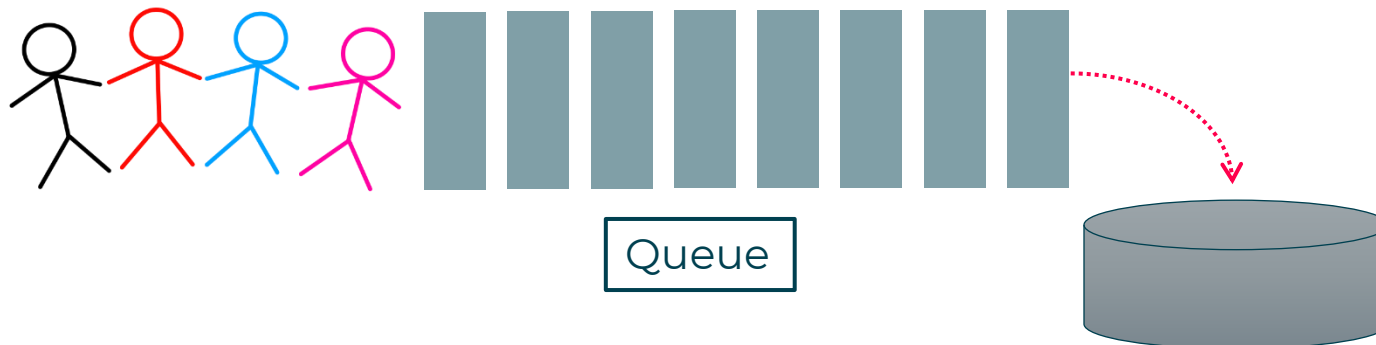
```
friends.set(1, "Susan");           // Susan replaces Sue
```

Collections framework Generic Types

- In package `java.util`

```
public class ArrayList<T> { ... }  
public class LinkedList<T> { ... }  
public class ArrayDeque<T> { ... } // stacks and Queues  
  
public class HashSet<T> { ... }  
public class TreeSet<T> { ... } // sorted  
  
public class HashMap <K,V> { ... }  
public class TreeMap <K,V> { ... }
```

Multiple type parameters allowed



Java: Queue – ArrayDeque<T> FIFO

- Provide both FIFO (Queue) & LIFO (Stack) behaviour

```
ArrayDeque<String> queue = new ArrayDeque<String>();  
queue.add("Dave");  
queue.add("Mike");  
queue.add("Linda");  
queue.add("Joe");  
  
while(!queue.isEmpty()) {  
    String item = queue.pop();  
    System.out.println(item);  
}
```



Dave
Mike
Linda
Joe

```
ArrayDeque<Car> queue= new ArrayDeque<>();  
  
queue.add(new Car("Ford"));  
queue.add(new Car("Honda"));  
  
for(Car car : queue) {  
    System.out.println(car.getModel());  
    queue.remove(car);  
}
```



Ford
Honda



Hands On Labs

Stack and Queue behaviours

- Part 1 – Using Lists
- Part 2 – Using Queues
- Please only do part 1 & 2

Usage of HashMap of key/value pairs

keys are
unique

```
HashMap<String, Car> hm = new HashMap<>();  
  
hm.put("Sam", new Car("Ford"));  
hm.put("Joe", new Car("BMW"));  
Car car = hm.get("Sam");  
  
System.out.println(car.getModel());
```

Ford

```
if(hm.containsKey("Bob"))  
    hm.put("Bob", new Car("Ferrari"));
```

Searching and
replacing an item

Usage of HashMap of key/value pairs

```
HashMap<String, Car> hm = new HashMap<>();
```

```
hm.put("Sam", new Car("Ford"));
```

```
hm.put("Joe", new Car("BMW"));
```

Joe drives a BMW
Sam drives a Ford

```
for (String key : hm.keySet()) {  
    System.out.printf("%s drives a %s\n", key, hm.get(key).getModel());  
}
```

```
for (Car car : hm.values()) {  
    System.out.println(car.getModel());  
}
```

BMW
Ford



Hands On Labs (Part 2)

- **Zoo Animals**
 - Using `HashMap<K, V>`

Review



Java 5.0 (2004) introduced generic types

- Improve performance and type safety, less casting
- `ArrayList<E>` & `HashMap<TKey, TValue>` widely used
- Stack and Queue behaviour exhibited by `ArrayDeque<E>`
- `java.util` package
- Map's are key value pairs (like a Dictionary)
 - Implemented by `HashMap` and `TreeMap`(sorted)
- `collections` utility class

Java Collections class

```
ArrayList<String> flavours = new ArrayList<>();  
..add add add  
flavours.sort();  
Collections.sort(flavours);  
Collections.reverse(flavours);
```



```
ArrayList<String> names = new ArrayList<String>(  
    Arrays.asList(new String []{"Dave", "Mike", "Linda", "Joe"}));  
ArrayList<String> devs = new ArrayList<String>(  
    Arrays.asList(new String[]{"Mike", "Joe"}));  
names.removeAll(devs);    [{"Dave", "Mike", "Linda", "Joe"}]);  
System.out.println(names);
```

[Dave, Linda]

```
ArrayList<Double> numbers = new ArrayList<>(  
    Arrays.asList(new Double[]{1.2, 4.8, 8.9, 3.7, 1.5}));  
double max = Collections.max(numbers);  
double min = Collections.min(numbers);  
Collections.swap(numbers, 3, 5);
```

8.9
1.2

// swap 2 elements