CPIT-252
Software Design Patterns

Java Collections and Generics

Last updated: 20/03/2022

Khalid Alharbi, Ph.D.

Introduction

- We often store our data in variables and create instances (objects) of our classes.
- We may use simple data structures such as arrays to store our values.
- We may also arrange elements in slightly more complex data structures such as multi-dimensional arrays to represent elements in rows and columns.
- We may work with various type of data where these options are not appropriate solutions to our problems.

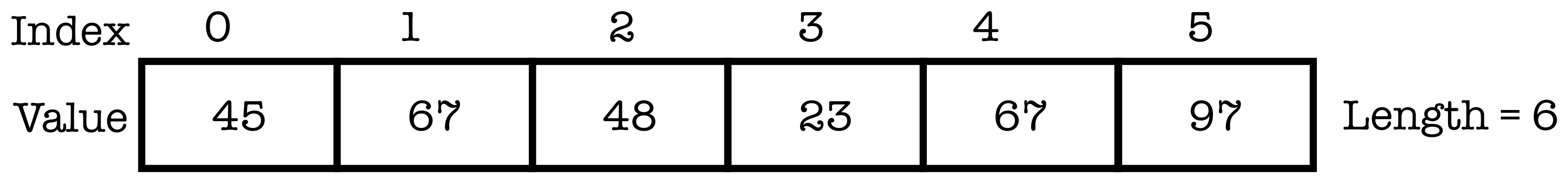
Introduction (cont.)

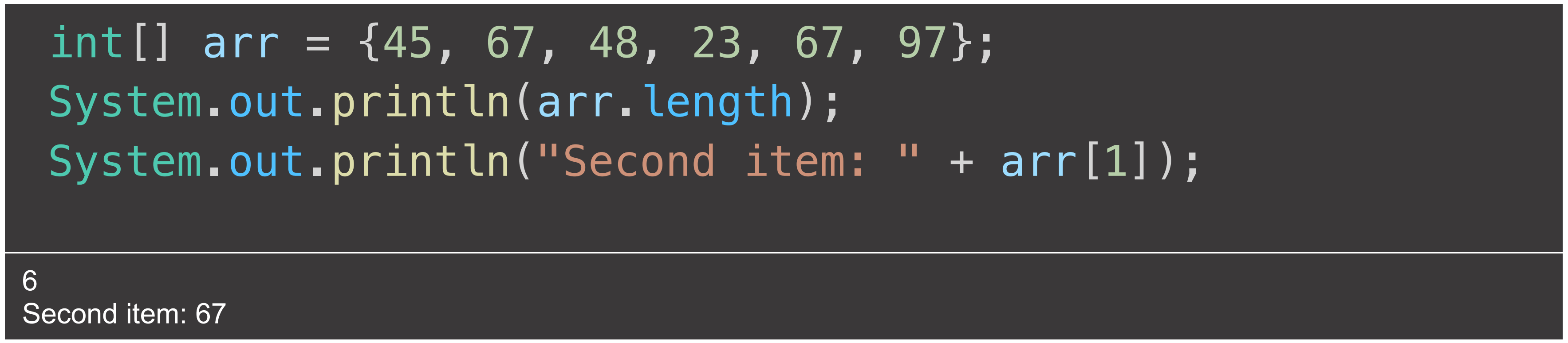
```
String username = "ali990";
Student s = new Student(2200990, "Ali", 3.90);
String[] courseTaken = new String[42];
// 2-d array initialized with values
int[][] courseYear = { { 252, 2022 }, { 405, 2022 }, { 305, 2021 } };
```

A program that looks like this may be a start, but we often deal with various types of data that need to be stored in a collection with efficient flexible operations on large amounts of data.

Array

- An **Array** is a sequence of elements or values of the same type.
- Each element is stored at a zero-based location called **index**.
- The number of elements in an array is called **length.**





```java
int[] arr = {45, 67, 48, 23, 67, 97};
System.out.println(arr.length);
System.out.println("Second item: " + arr[1]);
```

```
6
Second item: 67
```

Multidimensional Array

- A multidimensional arrays is a collection of elements or values of the same type.
- In two-dimensional arrays, elements are laid out in a rectangular grid with rows and columns.

Column	0	1	2	3	4	5
Row 0	45	67	48	23	67	97
Row 1	17	9	28	0	11	32

```java
int[][] arr = { { 45, 67, 48, 23, 67, 97 }, { 17, 9, 28, 0, 11, 32 } };
int[][] arr2 = new int[2][6]; // 2 rows 6 columns initialized with zeros.
System.out.println(arr[0][0]); // row 0, column 0 => 45
System.out.println(arr[0][1]); // row 0, column 1 => 67
```

Limitations of arrays

- The use of arrays will inevitably lead into problems due to their inherent limitations:
  - Arrays are fixed in size.
  - Arrays can not store data of various data types.
  - No easy duplicate removal
  - No default and optimized search and sort feature
- As our program continues to grow and the amount of data increases, we will need bigger, more efficient and flexible data structures.
- **Data structure** is a way to organize data in a cohesive unit that enables efficient access and modification.

The Java Collection Framework

The Java Collection API was introduced to solve the problem of storing various types of data inside container data structures with an efficient and flexible access.

We will look at the available data structures in the Java collection framework: Set, List, Queue, Deque, and Map. Finally, we will see how generics make collections powerful.

Collections

- **Collections** are fundamental data structures for storing and accessing data.
- A **collection** is an object that represent and manipulate a group of objects called elements.
- Some collections allow duplicate elements and others do not. Some are ordered and others are unordered.
- Learning how to use collections will help improve the efficiency and performance of your program as well as increase your productivity since you do not need to implement these powerful data structures and reinvent the wheel.

The Java Collection Framework

- The Java Collection Framework is a unified library of classes and interfaces for working with collection objects.
- It provides methods for *adding*, *removing* and *searching* for a particular element within a collection of elements.
- The collection framework defines a handful of interfaces in the `java.util` package.
- These interfaces are divided into two groups:

    1) `java.util.Collection`: used for containers that hold elements.

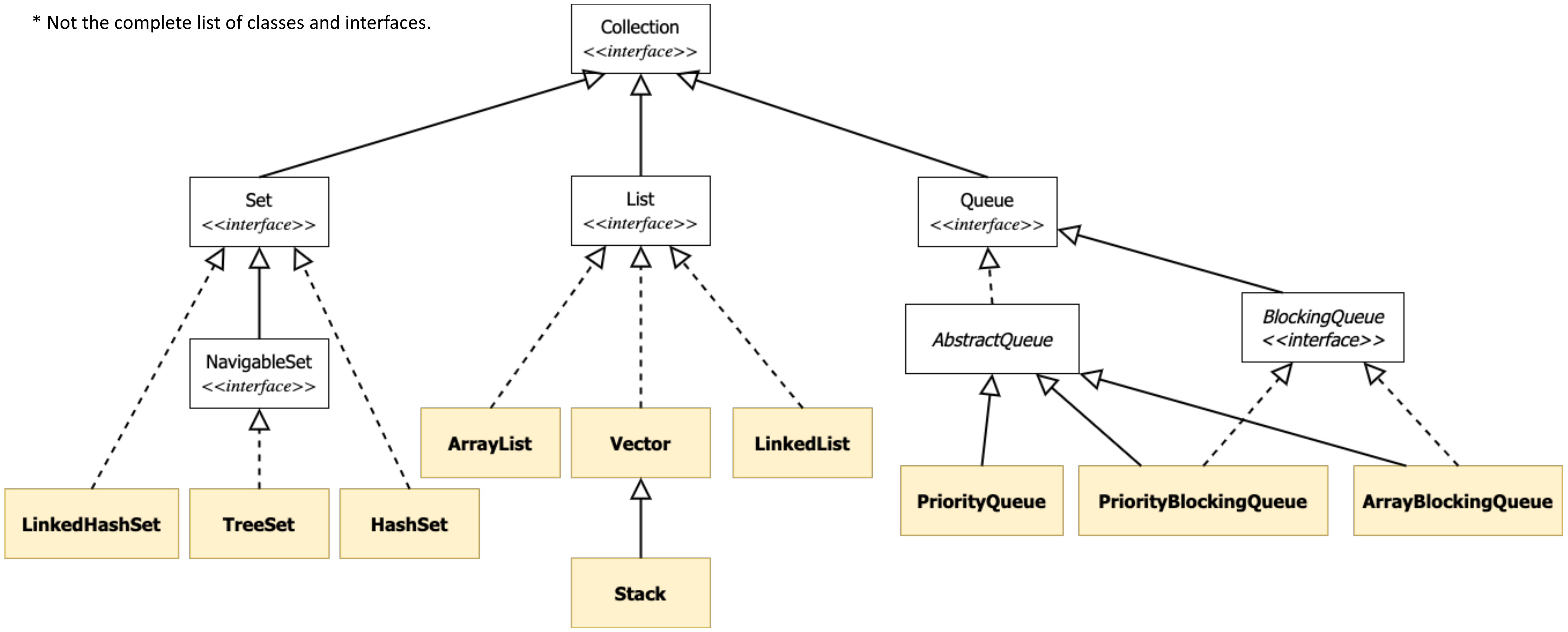
    2) `java.util.Map`: used for key/value pairs.

Part 1: `java.util.Collection` interface

The Collection interface is used for containers that hold elements.

The Collection interface* (I)

* Not the complete list of classes and interfaces.



The Collection interface (II)

- The Java collection framework uses three main interfaces: Set, List, and Queue.
- Each interface is implemented by concrete classes in multiple ways:

- `HashSet`, `TreeSet` and `LinkedHashSet` implement `Set`
- `ArrayList` and `LinkedList` implement `List`
- `LinkedList` , `ArrayDeque` implement `Queue`

The Collection interface (III)

- It defines common operations to all collections and describes what it does but not how it does it. Examples:
- Add an element to the collection:
  - `public boolean add(element)`
- Remove an element from a collection:
  - `public boolean remove(element)`
- Check if an element is in the collection:
  - `public boolean contains(element)`
- Iterate or loop through all elements using an *Iterator* object
  - `public Iterator iterator()`
- Get the size of the collection:
  - `public int size()`

Constructing a Collection (I)

```
Interface<Type> name = new Class<Type>();
```

- The type of the collection's elements is specified between < and >.
- The type must be a class type (e.g., Integer, String, etc.) and can't be a primitive data type (e.g. int, boolean, etc.)
- This is called a **Generic** class, which allows a collection to store elements of different types. Examples:

```
Set<String> mySet = new HashSet<String>();

List<Integer> myList = new ArrayList<Integer>();

Queue<Student> myQueue = new LinkedList<Student>();
```

Constructing a Collection (II)

- The Java Collections framework makes heavy use of interfaces to describe abstract data types:
  - `Set`, `List`, `Queue`, `Deque` and `Map`
- We declare variables using interface types with elements of object data types.
  - Why interfaces? Because we can use a different implementation later without requiring significant changes to existing code. This means that we can add, change and remove elements in the same way.

```
List<Integer> myList = new ArrayList<Integer>();

myList = new LinkedList<Integer>();
```

Autoboxing and Unboxing

- Unlike arrays, collections can not store primitives directly and only store objects.
  - The Java compiler will convert primitives into their corresponding objects through auto-boxing and back to primitives through unboxing.

```java
List<Integer> myList = new ArrayList<Integer>();

myList.add(23);             // autoboxing  (int -> Integer)

int a = myList.get(0);      // unboxing    (Integer -> int)
```

Collections

- We will look at the following data structures in the Java collection framework:

1. Set:
   - HashSet
   - TreeSet
   - LinkedHashSet
2. List:
   - ArrayList
   - LinkedList
   - Vector
3. Stack
4. Queue
5. Deque

Set (I)

- A **Set** contains an unordered collection of unique elements.
- It models the mathematical set model where duplicate elements are not allowed.
- A Set has no notion of position of stored elements within the collection.
- If you try to add an element that already exists in a Set, the add() method will return false.

Set (II)

The Set interface has three concrete implementations:

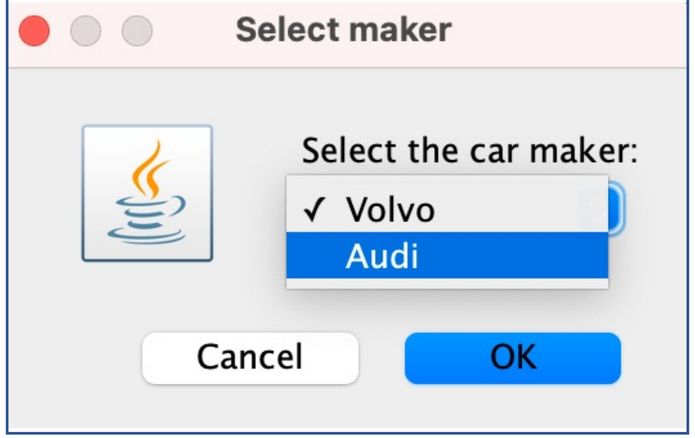
1. The **HashSet** class implements the Set interface and stores elements in a hash table.
   - Best for performance but order of elements is not guaranteed.
2. The **TreeSet** class implements the Set interface and stores an ordered set of elements in a TreeMap data structure.
   - Slower than a HashSet but it orders its elements based on their values and can find the closest match for a target, greater than or lesser than a given search target.
3. The **LinkedHashSet** class implements the Set interface and stores elements in a hash table with a linked list running through it
   - It orders its elements based on the insertion-order.

Set Example:

```java
public static void main(String[] args) {
        Set<String> mySet = new HashSet<>();
        mySet.add("Audi");
        mySet.add("Volvo");
        mySet.add("Audi");
        System.out.println(mySet);
        // show the set in a dropdown menu
        JOptionPane.showInputDialog(null,
                "Select the car maker:","Select maker",
                 JOptionPane.QUESTION_MESSAGE,null,
                 mySet.toArray(), null);
   }
```

Output:

```
[Volvo, Audi]
```



Demo

List (I)

- A **`List`** is an ordered sequence of elements.
- Unlike a `Set`, a `List` may contain duplicate elements.
- It is like an array but with a variable length and methods for manipulating the position of elements in the list.
- Unlike arrays, a list is a collection, so we can not store primitives directly and only store objects.
  - The Java compiler will convert primitives into their corresponding objects through auto-boxing.

List (II)

- In Java, the `List` interface has a handful concrete implementations:

1. **`ArrayList`** class implements the `List` interface using a resizable/growable array.
   - It’s often faster for adding/removing at the end and better-performing implementation in most scenarios.
2. **`LinkedList`** class implements the `List` interface using a doubly LinkedList.
   - It’s often faster for adding/removing at the beginning and middle and has better-performing implementation in certain scenarios.

List (III)

3. **`Vector`** class implements the *`List`* interface using a resizable or growable array.
   - Both the *Vector* class and the *ArrayList* class are almost equivalent. The main difference is that access to a *Vector* is synchronized (thread-safe) whereas it is not synchronized for an *ArrayList*.
   - If a thread-safe implementation is not needed, it is recommended to use *ArrayList* in place of *Vector*.
   - You may also use the *Collections.synchronizedList* function with an *ArrayList* to create a synchronized list and get the equivalent of a *Vector*.

More on List Implementations: ArrayList

- **`ArrayList`** is built using an internal uninitialized *array* and a *size* field to keep track of its capacity.
  - If an element is added, it first verifies whether it has enough capacity in the array to store new element or not. If there's enough capacity, the array is increased by **50%** to create a larger array and copy the original array into it.

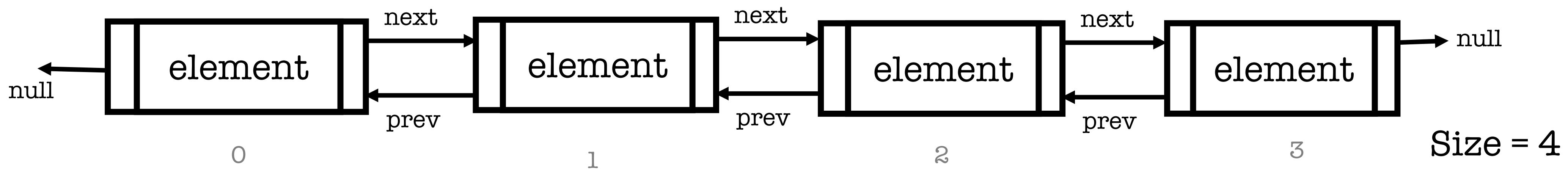
Index	0	1	2	3	4	5	
Value	obj1	obj2	0	0	0	0	Size = 2

- The initial capacity can be defined upon creating the ArrayList:

```
    List<Student> myList = new ArrayList<Student>(20);
```

More on List Implementations: LinkedList

- **`LinkedList`** is built using a doubly LinkedList where small node objects keep links to the next and previous node objects.
- These node objects form a chain with pointers to the next and previous node elements.



More on List Implementations: Vector

- **`Vector`** is quite similar to the *ArrayList*’s implementation, which implements a growable *array* of objects with a *size* field to keep track of its capacity, but with additional thread-safety and synchronized implementation.
  - If there’s enough capacity, the array is increased by **100%** to create a larger array and copy the original array into it.
  - Vector is a legacy class since JDK 1.0 and is considered slower due to synchronization.

Index	0	1	2	3	4	5	
Value	obj1	obj2	0	0	0	0	Size = 2

- The initial capacity can be optionally set upon creating the *Vector*:

```
      List<String> myList = new Vector<String>(20);
```

`List` Methods

Method	
public boolean add(E element)	Adds the specified element to the end of the list
public void add(int index, E element)	Adds the specified element at the given position.
public void remove(int index)	Removes the element at the specified position
public E get(int index)	Returns the element at the given position
public E set(int index , E element)	Changes the element at the given position
public void clear()	Removes all elements from the list.
public Iterator<E> iterator()	Returns an object used to iterate through the elements of the list.

// Note: **E** is the parameterized element type that is specified when creating the List. For example, below are examples where E is of a String class type and an Integer class type:

```
List<String> myStringList = new ArrayList<String>();
List<Integer> myIntegerList = new ArrayList<Integer>();
```

List Example: **ArrayList** implementation

```java
List<String> myList = new ArrayList<String>();
myList.add("Ahmed");
myList.add("Ali");
myList.add("Abdullah");
System.out.println(String.format("Content: %s, Size: %d", myList, myList.size()));
myList.add(1,"Khalid");
System.out.println(String.format("Content: %s, Size: %d", myList, myList.size()));
```

```
Content: [Ahmed, Ali, Abdullah], Size: 3
Content: [Ahmed, Khalid, Ali, Abdullah], Size: 4
```

Demo

List Example: **LinkedList** implementation

```
List<String> myList = new LinkedList<>();
myList.add("Ahmed");
myList.add("Ali");
myList.add("Abdullah");
System.out.println(String.format("Content: %s, Size: %d", myList, myList.size()));
myList.add(1,"Khalid");
myList.remove("Abdullah");
System.out.println(String.format("Content: %s, Size: %d", myList, myList.size()));
```

```
Content: [Ahmed, Ali, Abdullah], Size: 3
Content: [Ahmed, Khalid, Ali], Size: 3
```

Demo

List Example: **Vector** implementation

```java
List<String> myList = new Vector<>();
myList.add("Ahmed");
myList.add("Ali");
myList.add("Abdullah");
System.out.println(String.format("Content: %s, Size: %d", myList, myList.size()));
myList.add(1,"Khalid");
myList.remove("Abdullah");
System.out.println(String.format("Content: %s, Size: %d", myList, myList.size()));
```

```
Content: [Ahmed, Ali, Abdullah], Size: 3
Content: [Ahmed, Khalid, Ali], Size: 3
```

Demo

Stack (I)

- The **`Stack`** class extends the *`Vector`* class to represent a last-in-first-out (LIFO) stack of objects with five operations: *push(E item), pop(), peek(), search(Object o)* and *empty()*.
  - It retrieves elements in the reverse order they were added in.
  - The Stack class does not provide a complete and consistent set of LIFO stack operations.
  - It’s a less powerful as a collection implementation but is optimized to perform LIFO operations very quickly.
  - Stacks are often implemented using an array
  - The *`Deque`* interface and its implementations are preferred over the *`Stack`* class (seen later).

Stack Example (I)

- Recall that the Java compiler will convert primitives into their corresponding objects through auto-boxing and back to primitives through unboxing.

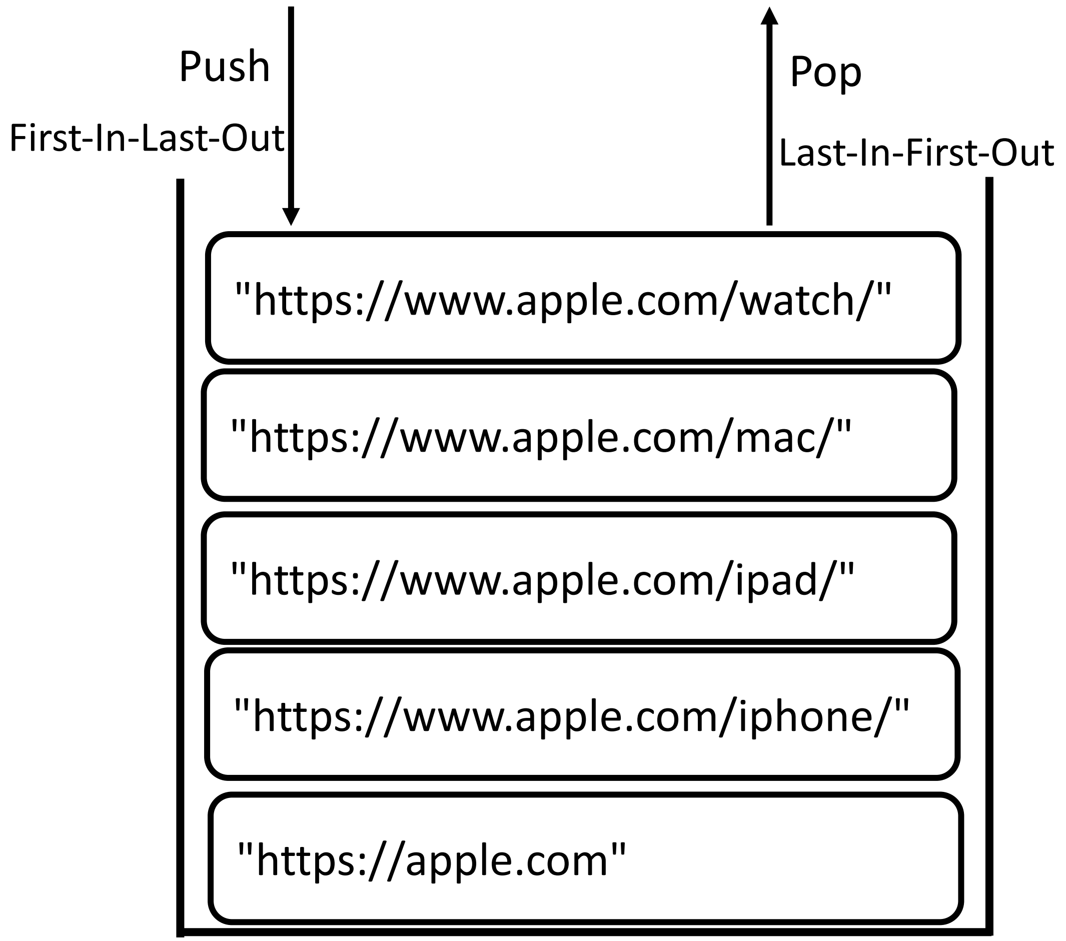
```
Stack<Integer> myStack = new Stack<Integer>();
myStack.push(20);          // autoboxing  (int -> Integer)
myStack.push(40);
myStack.push(60);
int a = myStack.pop();     // unboxing    (Integer -> int)
System.out.println(a);
System.out.println(myStack);
```

```
60
[20, 40]
```

Demo

More Stack Applications

- Examples of First-In-First-Out (FILO) applications:
  - The “Undo” operation in a text editor.
  - The “Back” button to return to previous browsing history.


Push
First-In-Last-Out
Pop
Last-In-First-Out
"https://www.apple.com/watch/"
"https://www.apple.com/mac/"
"https://www.apple.com/ipad/"
"https://www.apple.com/iphone/"
"https://apple.com"

Stack Example: Browsing History

```java
Stack<String> history = new Stack<String>();
history.push("https://apple.com");
history.push("https://www.apple.com/iphone/");
history.push("https://www.apple.com/ipad/");
history.push("https://www.apple.com/mac/");
history.push("https://www.apple.com/watch/");
while(!history.empty()){
    System.out.println(history.pop());
}
```

```
https://www.apple.com/watch/
https://www.apple.com/mac/
https://www.apple.com/ipad/
https://www.apple.com/iphone/
https://apple.com
```

Demo

Queue

- A *Queue* typically, but not always, retrieves elements in a FIFO (first-in-first-out) order.
  - There are special queue implementations such as a priority queue where elements are ordered according to their natural ordering or a Comparator method.
  - A queue may be bounded or unbounded.
    - A bounded queue implementation uses a fixed-sized array to hold the elements, where capacity can't be changed.
- There are many concrete implementations of the *Queue* interface:
  - LinkedList
  - PriorityQueue
  - SynchronousQueue
  - ArrayBlockingQueue
  - ConcurrentLinkedQueue
  - DelayQueue

Queue Example

```
Queue<Integer> ordersQueue = new LinkedList();
ordersQueue.add(3);
ordersQueue.add(1);
ordersQueue.add(5);

while(!ordersQueue.isEmpty()){
    System.out.println(ordersQueue.remove());
}
```

```
3
1
5
```

Demo

Deque

- A Deque is a queue that supports element insertion and removal at both ends.
- The name deque is short for "double ended queue" and is pronounced "deck".
- Most *Deque* implementations, but not always, are unbounded with no fixed limits on the number of elements they may contain
- There are many concrete implementations of the *Queue* interface:
    - ArrayDeque
    - LinkedList
    - LinkedBlockingDeque
    - ConcurrentLinkedDeque

Deque Example

```java
Deque<Integer> ordersDeque = new ArrayDeque();
ordersDeque.add(3);
ordersDeque.add(1);
ordersDeque.add(5);
ordersDeque.addFirst(0);
ordersDeque.addLast(7);

while(!ordersDeque.isEmpty()){
    System.out.println(ordersDeque.remove());
}
```

```
0
3
1
5
7
```

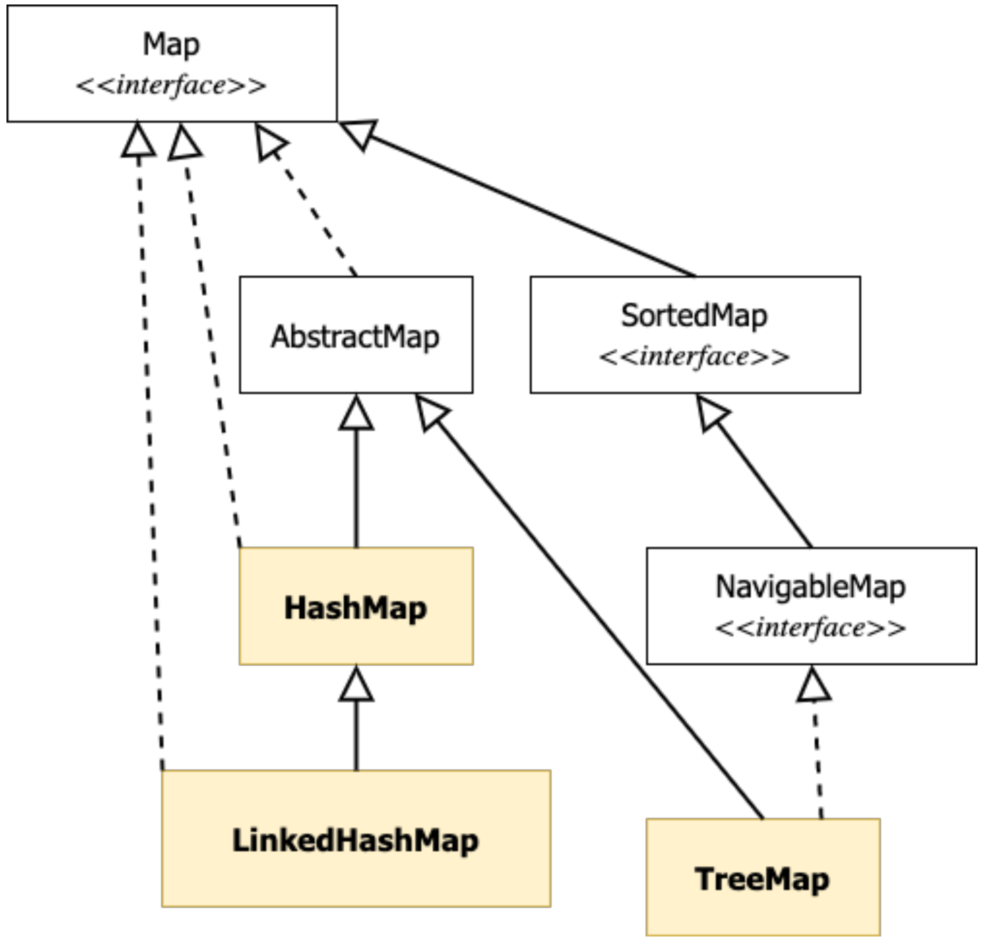
Demo

Part 2: `java.util.Map` interface

The Map interface is used for storing key/value pairs.

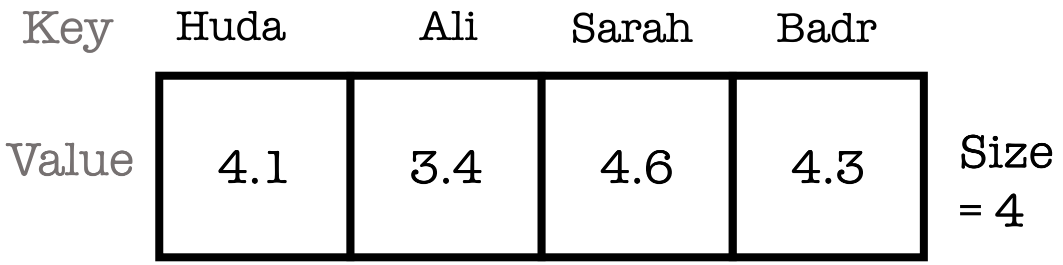
The Map interface*

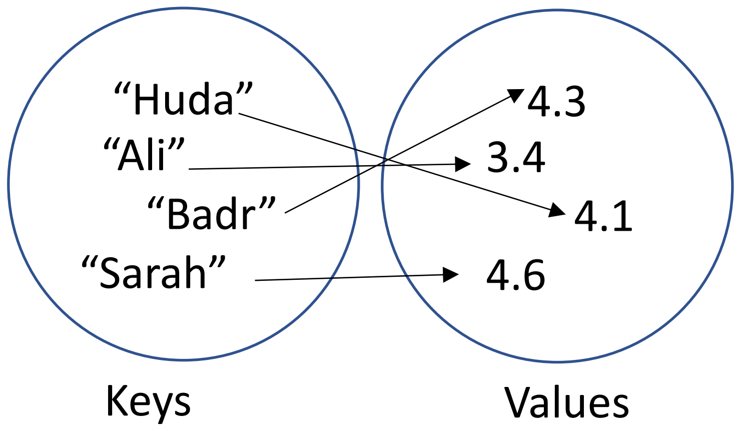
* Not the complete list of classes and interfaces.



Map (I)

- A *`Map`* can be thought of as an array with an index that does not have to be an `int`. It's also known as "dictionary" or "associative array."
- A Map stores key-value pairs acting like a cache or a minimalist database.
- Maps cannot contain duplicate keys; each key is unique and maps to a single value.





Map (II)

- With generics, a Map type is parameterized with two types: one for the keys and one for the values.
- A Map has the following basic operations:
    - put(key, value): Creates a mapping from the key to the specified value.
    - get(key): Returns the value by the given key.
    - remove(key): Removes the mapping for the given key.

```
Map<String, Date> m = new HashMap<String, Date>();

m.put("request-2345-ghy", new Date());
```

Map (III)

The *Map* interface has three concrete class implementations (analogous to the *Set* Interface):

1. **`HashMap`** stores an *unordered* set of key-value pairs in a hash table.
   - HashMap does not maintain order but is considered faster and with O(1) for the basic get() and put() operations.
2. **`TreeMap`** stores an *ordered* set of key-value pairs (according to the natural ordering of its keys) in a TreeMap data structure.
3. **`LinkedHashMap`** stores an *ordered* set of key-value pairs (according the insertion order) in a hash table with a doubly-linked list running through it.

`Map` Methods

Method	
`public <V> put(K key, V value)`	Adds a mapping from the given key to the given value.
`public <V> get(Object key)`	Returns the value mapped to the given key or null if not found.
`public boolean containsKey(K key)`	Returns true if there's a mapping for the specified key.
`public <V> remove(Object key)`	Removes the mapping for a key if it exists
`Set<K> keySet()`	Returns the unique set of keys in the map.
`Collection<V> values()`	Returns the values contained in the map.
`public void clear()`	Removes all key/value pairs from the map.

// Note: **K** and **V** are the parameterized element type that are specified when creating the Map. For example, below are examples where **K** is of a String class type and **V** is an Integer class type:

```
Map<String, Integer> myMap = new HashMap<String, Integer>();
```

Map Example

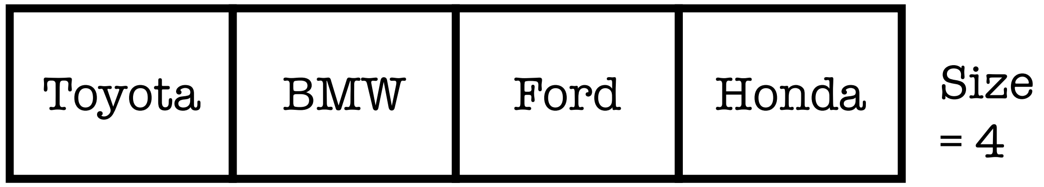
```java
Map<String, Integer> vehicles = new HashMap<String, Integer> ();
vehicles.put("BMW", 5);
vehicles.put("Audi", 4);
vehicles.put("Ford", 10);

for(String v: vehicles.keySet()){
    System.out.println(String.format("We have %d %s vehicles", vehicles.get(v), v));
}
```

```
We have 4 Audi vehicles
We have 10 Ford vehicles
We have 5 BMW vehicles
```

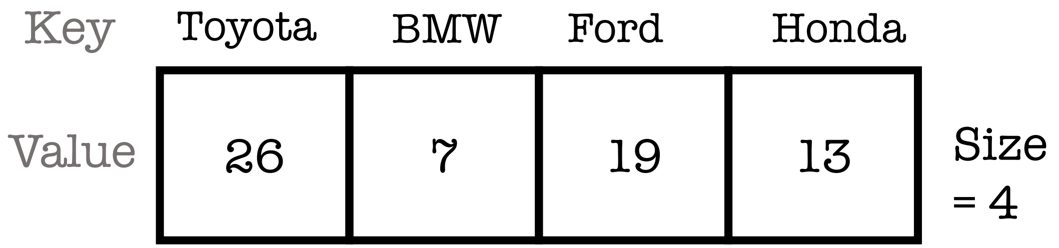
Demo

One more thing on Maps: Maps vs. Sets

- A `Set` contains a unique elements of values and it is more like a map from elements to boolean values (true). While a `Map`, is a map from unique set of keys to values.
- **`Set`**: Is "Toyota" in the set? *(true or false)*



- **`Map`**: How many "Toyota" cars do we have? *(mapping a key to a key)*



Part 3: Generics

Work with a general data type

Let’s answer the following questions:

- How do we store data without **casting**
- Any takers?

Generics: Introduction (I)

- In OOP, **polymorphism** means that objects can take many forms and are mostly interchangeable.
- **Polymorphism** occurs when a parent class reference is used to refer to a child class object.
  - Example: If *Truck* is a subtype of *Vehicle*, then we can use an object of type *Truck* anywhere that we expect something of type *Vehicle*.

```java
public interface Vehicle{}
public class Truck extends Vehicle{}
Vehicle v = new Truck();
```

Generics: Introduction (II)

- Generics enable creating classes and methods that work with any object data types.
- Generics are also known as “parameterized types” or “templates”.
- With Generics, we can write a function that handles parameters of different data types without depending on their actual data types.
  - Such a function is called *generic* function, or it takes a generic data type.
- Generics allow us to write general or flexible code without sacrificing the static type safety.
- Generics are heavily used in Java *Collections.*
  - Java Collections and Maps can hold just about any object type without the need to do data type casting to convert one type into another.

Problems without Generics?

- In Java, an object is a child of the general type **`java.lang.Object`**.
  - If the *`ArrayList`* takes a `java.lang.Object`, it can accept any object, which is error-prone, and the return value must be casted/converted, which can also lead to run-time errors and unexpected behavior.

```java
ArrayList myList = new ArrayList();
myList.add("BMW");
myList.add(90); // This is bad, but compiler will allow it (with a warning)!
myList.add("Audi");

String bmw = (String) myList.get(0); // We must cast from Object to String
int quantity = (Integer) myList.get(1); // We must cast from Object to Integer
System.out.println(bmw); // prints BMW
System.out.println(quantity); //prints 90
```

Can we do better without Generics? (I)

- Is there a way to make an ArrayList that only accepts String into the list? Can we get rid of the casting? Can we do better?
- What if we override the `add()` method in a subclass that only accept String objects?
  - Unfortunately, doing so will not override anything!! Instead, it will create another overloaded method. We can add any non-String data type to our list.

```java
public void add( Object o ) { ... } // still here
public void add( String s ) { ... } // overloaded method
```

Can we do better without Generics? (II)

- What if we take a bigger approach and write our own `StringList` class that does not extend `ArrayList` but delegates to it.
  - Unfortunately, doing so will make our new `StringList` class not a *List*, so it will not work with any Java Collection methods (e.g., `sort()`, `addAll()`)

```java
class StringList{
    ArrayList l;
    public StringList(ArrayList l){
        this.l = l;
    }
    public boolean add(String item){
        return l.add(item);
    }
}
```

Generics in Action

- The class *`java.util.List`* is a generic class and is declared as:

```
class List<E>{
    public void add(E element){ //TODO: Add to the list }
    public E get(int i) { //TODO: Get from the list }
}
```

- The identifier *`E`* between the angle brackets (<>) is a type parameter.
  - It indicates the class *`List`* is generic and requires a data type as an argument.
  - The letter *`E`* is chosen as a naming convention, but it can be anything else.
- Example: Declaring a variable using the generic type List with a type parameter of String: `List<String> myList;`

How do generics solve these problems?

- If the **`ArrayList`** was made generics, then it would take only the parameterized element type and threw a compile time error in any other case (type-safety) and no need for casting (flexibility).

```
ArrayList<String> myList = new ArrayList<String>();
myList.add("BMW");
// myList.add(90); This is wrong and will result in compiler time error
myList.add("Audi");

String bmw = myList.get(0); // Casting is not needed
System.out.println(bmw); // prints BMW
```

Generics: Conclusion

- Generics enable classes and interfaces to be used as parameter types when defining classes, interfaces and methods.
- Type parameters allow us to re-use the same classes with different data types.
- Generics allow stronger type-safety at compile time.
  - Recall that fixing compile-time errors is much easier than fixing runtime errors.
- Generics eliminate the need for casting/type conversion.
- Generics enable us to work on collections of different types.

Wrapping up

- Collections and generics are powerful abstractions in the Java programming language.
- Collections allow for flexible and efficient containers that hold other objects.
- The `java.util.Collection` interface has three main child interfaces:
  - `Set`: *A collection of unique elements*
  - `List`: A collection of elements with a specific order.
  - `Queue:` A collection of elements retrieved in specific fashion (e.g., first-in-first-out or based a specific priority value).
- The `java.util.Stack` interface A collection of elements retrieved in specific manner (first-in-last-out).
- The `java.util.Map` interface represents a collection of key/value pairs.