Data Structures

Agenda

- Strings
- JDK collections
- JDK collection utilities
- Java collection libraries

Strings

Overview

- We already discussed a data structure provided by the Java programming language: arrays

- While strings are considered more of a data type there are backed by an array and due to their importance we are going to cover them extensively

Strings

- Strings can be easily created using string literals

```
String text = "Java Academy"
```

- String literals are stored in a string pool area of the JVM heap

- Strings are immutable (i.e. cannot be modified but rather a new String is created instead)

String concatenation

- String concatenation is the process of combining multiple strings into one

```
String text = "Java" + "Academy"
```

- Since strings are immutable string concatenation needs to be avoided in some cases (as a lot of new String objects might be created)

```
String text = "";
for(int i = 0; i < words.length; i++) {
        // this needs to be avoided !
        text += words[i];
}
```

String concatenation

- The preferred way to concatenate Strings is to use the **java.lang.StringBuilder** or **java.lang.StringBuffer** utilities

```
StringBuilder builder = new StringBuilder("");
for(int i = 0; i < words.length; i++) {
	builder.append(words[i]);
}
String text = builder.toString();
```

- StringBuffer is the thread-safe equivalent of StringBuilder

String methods

- The **java.lang.String** methods include:

charAt	character at given position
concat	concatenates the string with another one (the same as with + operator)
indexOf	returns the index of a first occurrence of a character or string
lastIndexOf	Similar to **indexOf** but returns index of last occurrence
intern	intern: returns the same string from the string pool
isEmpty	Returns **true** if the string is empty
isBlank	Returns **true** if the string is empty or contains only whitespaces (since JDK 11)

String methods

- The **java.lang.String** methods include:

chars	Returns a stream of characters
lines	Returns a stream of the lines of the String
matches	Return **true** if the String matches a regular expression
repeat	Returns a string that is a repetition of the current String n-times
replace	Replaces a character or a string with another string
replaceAll	Replaces all matches of a regular expression with another string
replaceFirst	Replaces the first occurrence of a String with a regular expression

String methods

- The **java.lang.String** methods include:

split	Splits a string based on a given regular expression
startsWith	Returns **true** if the given string starts with another string
strip	Returns a string without leading and trailing whitespaces
substring	Returns a substring starting from a given position and ending at another position (optional)
toCharArray	Returns an underlying char array corresponding to the string (internal representation if a byte array as of JDK 9 introduced as **compact strings**)

String methods

- The **java.lang.String** methods include:

toLowerCase	Returns the lowercase equivalent of the string
toUpperCase	Returns the uppercase equivalent of the string
format	Formats the string according to a given format string
join	Combines an array of strings into a single one using a given delimiter
valueOf	A set of methods used to return a string from a primitive type, char array or an Object

The plus operator can be used to convert a number to a string

```
int num = 10;
String text = "" + num;
```

Converting strings to numbers

- To convert a string a number on the parseXXX methods from the corresponding wrapper types can be used

```
Float first = Float.parseFloat("12.2f");
Double second = Double.parseDouble("12.2");
Integer third = Integer.parseInt("17");
```

JDK collections

Collections framework

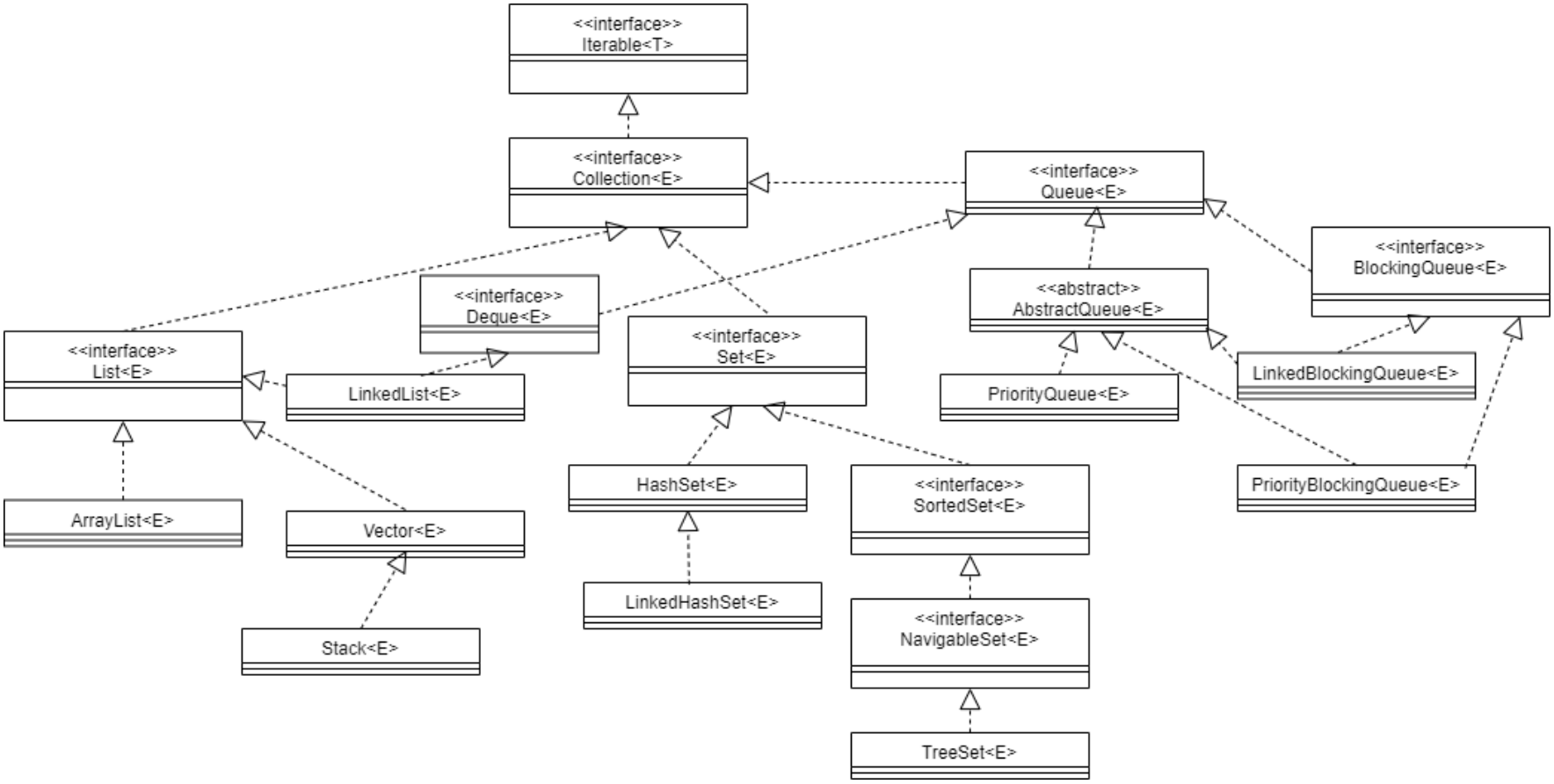
- A number of heavily used in practice data structures are provided by the Java collections framework

- The frameworks provides the following:

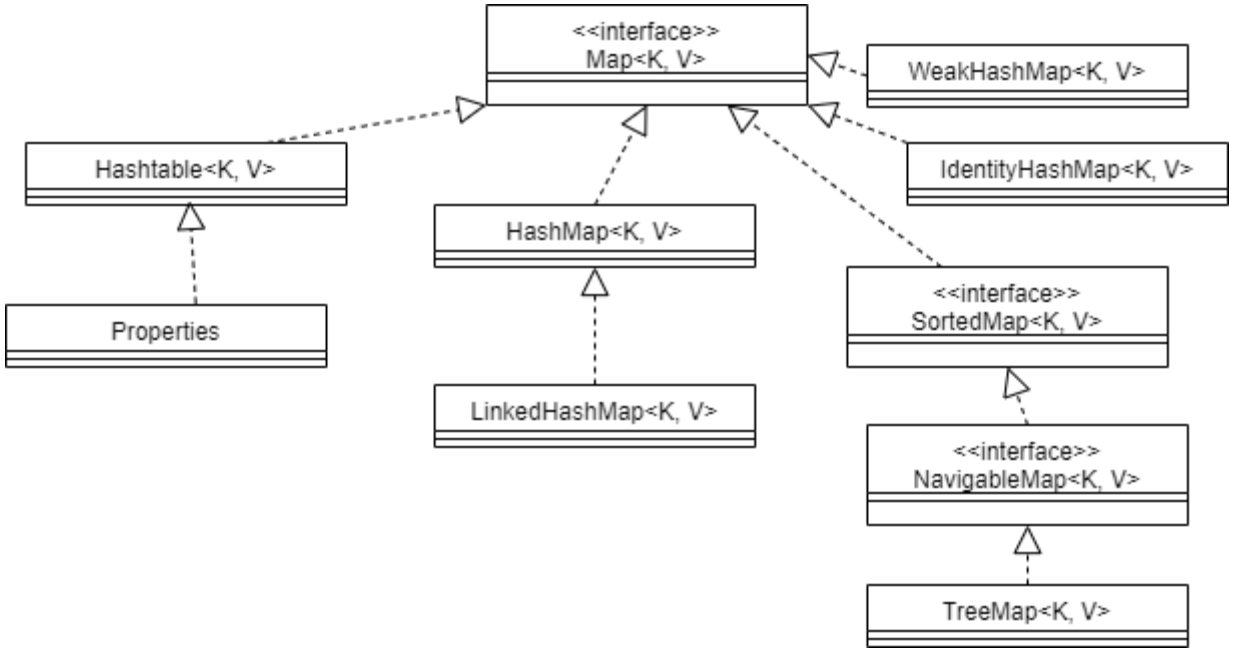
  - interfaces for interacting with the various data structures
  - Implementations of the interfaces
  - algorithms for manipulating the data structures

Using the collections framework is the preferred way to use a particular data structure rather than implementing one (unless the collections framework and no library provides a proper implementation)

Collections framework



Collections framework



Collections interface

- The collections interface provides common operations for all collections

size	Number of elements in the collection
isEmpty	Returns **true** if the collection is empty
contains/containsAll	Returns **true** if the collection contains the object(s)
iterator	Returns an iterator over the elements of the collection
toArray	Returns an array of the elements of the collection
add/addAll	Adds an element/elements to the collection
remove/removeAll	Removes an element/elements from the collection

Collections interface

- The collections interface provides common operations for all collections

toArray	Returns an array of the elements of the collection
clear	Empties the collection
equals	Checks if the collection is equivalent to another collection
hashCode	Returns the calculated hash code for the collection
stream/parallelStream	Returns a stream/parallel stream for the collection

We are going to cover streams when we talk about the JDK utilities

Iterators

- Iterators provide a mechanism to iterate over the elements of a collection
- They need to implement the **java.lang.Iterator<E>** interface
- The collections that can return an iterator need to implement the **java.lang.Iterable<T>** interface

Iterators

```
public interface Iterator<E> {

	boolean hasNext();

	E next();
}
```

```
LinkedList<String> list = new LinkedList<String>();
// add some elements to list ...
Iterator<String> iterator = list.iterator();
while(iterator.hasNext()) {
	String element = iterator.next();
	// do something with element ...
}
```

Traversing collections

- If collections implement the Iterable interface then the shorthand for-each operator can be used to traverse them

```
Collection<String> collection = ...
for(String element : collection) {
	// do something with element ...
}
```

- Regular for loop can be used to iterate over the elements of a collection as well

```
List<String> list = ...
for(int index = 0; index < collection.size(); index++) {
	String element = list.get(index);
	// do something with element ...
}
```

Lists

- Lists are an ordered collection of elements
- The two most widely used implementations are:
  - LinkedList: implementation of a linked list

```
LinkedList<String> elements = new LinkedList<String>();
```

  - ArrayList: list backed by an array (with a default capacity of 10)

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

LinkedList vs ArrayList

- Since list is used so often in practice it is essential to differentiate which implementation to use

- If faster search is needed ArrayList is preferred as it has constant access to an element by index (O(1)) and LinkedList has a linear access (O(n))

- Adding or removing elements from the beginning or end (head or tail) of a LinkedList it might be faster than with ArrayList

LinkedList vs ArrayList

- ArrayList requires more memory upfront and keep more memory even if not needed

- ArrayList is more preferable in the majority of scenarios where the size of the list is known upfront

Turns out that ArrayList is more practical to use for most scenarios than LinkedList

Sets

- A set is a data structure that contains unique elements (i.e. duplicates are not allowed)

- There are three main implementations of the Set interface:
  - HashSet

```
HashSet<String> cache = new HashSet<String>();
```

  - LinkedHashSet

```
LinkedHashSet<String> cache = new LinkedHashSet<String>();
```

  - TreeSet

```
TreeSet<String> cache = new TreeSet<String>();
```

HashSet

- HashSet is by far the most widely used type of set in practice
- Order of elements added to the set is not guaranteed
- Internally HashSet uses a HashMap

HashSet

- In order to work correctly the class of the objects being added to a HashSet needs to implement:
  - **hashCode**: inherited from the Object class, must return a unique hash code identifying the object.
  - **equals**: checks if the object is equal to another object. If there is an object within the set with the same hash code as returned by **hashCode** method then **equals** is called to further check the objects
- The same applies for HashMap when using custom objects as keys

Two different objects (where equals returns **false**) that have the same hash code (as returned by **hashCode** method) is called a **collision**

LinkedHashSet

- LinkedHashSet is an extension of HashSet
- Preserves the order of the elements in the set

As LinkedHashSet is an implementation of HashSet it still depends on **hashCode** and **equals** methods being implemented correctly !

TreeSet

- TreeSet works by comparing objects
- A **java.util.Comparator** may be supplied in the TreeSet constructor
- The class of objects being added to the TreeSet may implement the **java.lang.Comparable** interface

TreeSet

- Internally uses a TreeMap

- Ordering of elements depends on the Comparator or Comparable implementation being used

In terms of performance TreeSet is the least preferable option compared to HashSet and LinkedHashSet !

In terms of memory TreeSet also requires more memory than HashSet !

Comparator

```
public class TableComparator
	implements Comparator<Table> {

	@Override
	public int compare(Table first, Table second) {
		return first.getSize() > second.getSize();
	}
}
```

```
TableComparator comparator = new TableComparator();
TreeSet<Table> treeSet = new TreeSet<Table>(comparator);
```

Comparable

```
public class Table implements Comparable<Table>{

	public int getSize() {
		return size;
	}

	@Override
	public int compareTo(Table other) {
		return getSize() > other.getSize();
	}
}
```

Queues

- Queues are a collection of elements whereby elements are added on end and removed from the other

- Additional operations are provided by the Queue interface:

  - **peek**: retrieves, but does not remove the head element of the queue

  - **poll**: retrieves and removes the head element of the queue

  - **element**: the same as **peek** but throws an exception if the queue is empty

Queues

- LinkedList implements the Queue interface and can behave as a Queue
- A special variant of a queue whereby elements can be added and removed on both ends is called deque
- The Deque interface extends the Queue interface and is implemented by LinkedList

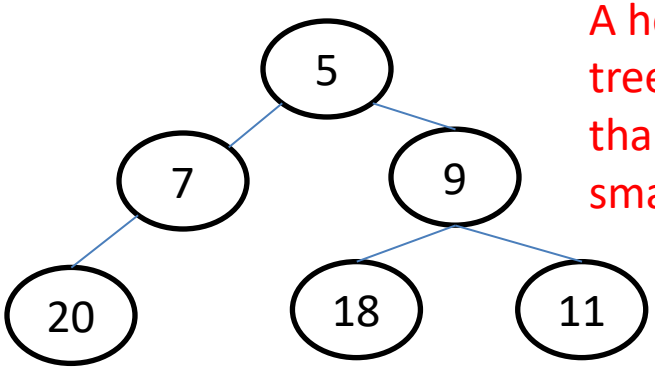
Deque

- The Deque interface provides additional operations:
  - addFirst/addLast
  - offerFirst/offerLast (same as addFirst/addLast but do not throw exception if deque capacity is reached)
  - removeFirst/removeLast
  - pollFirst/pollLast
  - getFirst/getLast
  - peekFirst/peekLast

PriorityQueue

- A priority queue orders elements when adding them to the collection and uses a Comparator for ordering

- PriorityQueue is a priority queue implementation that implements the Queue interface and is based on the heap data structure



A heap can be implemented as a binary tree where each child is larger or equal than the parent element (max heap) or smaller or equal (min heap)

The **java.util.concurrent** package provides concurrent queue implementations we are going to cover when we talk about concurrency

Maps

- A map provides the possibility to keep a collection of key-value pairs where duplicates are not possible

- The Map interface provides additional operations:

containsKey/containsValue	Checks is a given key/value is already present in the map
get	Returns the value corresponding to a given key
put	Puts a key-value pair in the map
remove	Removes a key-value pair from the map

Maps

- A map provides the possibility to keep a collection of key-value pairs where duplicates are not possible

- The Map interface provides additional operations:

putAll	Puts all key-value pairs from another map
keySet	Returns a set of the keys of the map
entrySet	Returns a collection of key-value pairs of the map

It is more preferable and performant to use the **entrySet** method in order to iterate over the key-value pairs of the map

Maps

- A map entry is represented by the inner Map.Entry<K,V> interface that holds the key-value pair

- There are three map implementations that have the same characteristics as they corresponding Set counterparts:

  - HashMap
  - LinkedHashMap
  - TreeMap

A number of default methods are added to the Map interface

HashMap is the most widely used Map implementation and preferred in terms of performance

JDK collection utilities

Collection utilities

- There are a number of heavily used in practice operations that can be used in practice
  - sorting the elements
  - searching for a particular element
  - finding minimum or maximum value
  - randomly shuffling the elements of the collection
- Many such operations are provided by the **java.util.Collections** class as static methods

Collection utilities

- Operations provided by **java.util.Collections** include:

sort	sorts a List using a merge sort algorithm
shuffle	randomly permutes the elements in a List
reverse	reverses the order of the elements in a list
swap	swaps the elements at specified positions in a list
replaceAll	replaces all occurrences of one specified value with another
copy	copies the source List into the destination List

Collection utilities

- Operations provided by **java.util.Collections** include:

binarySearch	searches for an element in an ordered List using the binary search algorithm
indexOfSubList	returns the index of the first sublist of one List that is equal to another
lastIndexOfSubList	returns the index of the last sublist of one List that is equal to another
min	returns the minimum element of a collection according to a comparator
max	returns the maximum element of a collection according to a comparator

Collection utilities

- Operations provided by **java.util.Collections** include:

unmodiableCollection unmodifiableList unmodifiableSet unmodifiableMap	returns an unmodifiable collection from a given collections
synchronizedCollection synchronizedList synchronizedSet synchronizedMap	returns a synchronized collection from a given collections
emptyList emptySet emptyMap	returns an empty collection

Java collections are not synchronized by default

Collection from an array

- The **java.util.Arrays** class provides an **asList** method that can create a list from an array of a given type

```
Arrays.asList(first, second, third);
```

- As of JDK 9 the various collection interfaces provide a default **of** method that can be used to return an unmodifiable collection from an array

```
Set.of(first, second, third);
```

```
List.of(first, second, third);
```

Java collection libraries

The Java collections framework

- The Java collection framework provides implementation of fundamental data structures
- It lacks however implementation of a multitude of other data structures (see https://en.wikipedia.org/wiki/List_of_data_structures)
- If a more specific data structure or operation is needed a third-party library can be used instead of implementing your own …

Persistent collections

- Some of the most widely used Java collection libraries at present include:
  - Apache Commons Collections
  - Eclipse collections (formerly Goldman Sachs collections)
  - Guava Collections (formerly Google Collections Library)
  - Persistent collections

Apache Commons collections

- Provides additional data structures such as Bag and BidiMap (bidirectional map)
- Provides a number of useful utilities under the CollectionUtils class such as:
  - collate: combines two lists into one
  - collect: transforms the elements of a collection
  - filter: filters the elements of a collection
  - isNotEmpty: checks if the elements of a collection are not empty
  - intersection: returns the intersection of two collections
  - union: returns the union of two collections

Eclipse collections

- Provides additional implementations of data structures such as immutable collections, primitive collections, bimaps, multimaps and bags

- Utility classes that also interoperate with the Java collections framework

Guava collections

- The Guava library provides a number of new collection types such as a Multiset, Multimap, BiMap, RangeSet and RangeMap

- Provides collection utilities that fulfill the Collections class provided by the Java collections framework and also utilities for working with Guava-provided collections

Persistent collections

- Provides persistent (versioned) and immutable implementation equivalent of the collections provided by the Java collections framework

HashTreePMap	analogous to HashMap
ConsPStack	analogous to LinkedList
TreePVector	analogous to ArrayList
HashTreePSet	analogous to HashSet
HashTreePBag	no JDK analogue (a bag allows duplicate elements)

Questions ?