GENERICS AND COLLECTIONS

CONTENT

- OCA/OCP11-1 recap
- Generic classes
- ArrayList, TreeSet, TreeMap, and ArrayDeque
- Comparator and comparable
- ForEach on streams and lists
- Method references with streams
- Lambda expressions II
- Stream API

ARRAY AND ARRAYLIST

- Array:
  - Contains objects or primitives
  - Fixed length
  - .length to get length
- ArrayList:
  - list of objects (cannot contain primitives)
  - Resizable
  - .size() to get length

CAN THIS BE DONE?

1. int[] test = {3,27,6};
2. List<int> list = Arrays.asList(test);
3. String[] array = {“hi”, “hey”, “bye”};
4. List<String> list2 = Arrays.asList(array);
5. list2.add(“hello”);
6. list2.set(1, “greetings”);
7. What is the value of array[1] now?

SEARCHING AND SORTING

- Array and ArrayList can be sorted
  - Arrays.sort(arrayName);
  - Collections.sort(listName);
- After sorting, binarySearch can be applied
  - Arrays.binarySearch(arrayName, searchValue)
  - Collections.binarySearch(listName, searchValue)
- Search returns:
  - Positive number or 0 >> index where value has been found
  - -1 >> not found, should have been the first in the array(list)
  - Smaller than -1 >> one smaller than the negated index of where value should have been inserted

EXERCISE: SEARCHING ARRAY

- Create an array with 10 000 random integers
- Do a binarysearch for 7
- How to interpret this result?
  What can you conclude?

WRAPPER CLASSES

- Each primitive has corresponding wrapper class
- Convert primitives to objects
- Autoboxing: Automatically converting primitive to corresponding wrapper class
- Unboxing: Automatically converting wrapper class to primitive

- Double d = new Double(5.0);

OVERVIEW WRAPPER CLASSES

Primitive	Wrapper class
boolean	Boolean
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character

EXERCISE: WHERE DO WE SEE AUTOBOXING AND UNBOXING?

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

1. list.add(6);
2. list.add(new Integer(1));
3. list.add(new Integer(4));
4. list.remove(1);
5. list.remove(6);
6. int x = list.get(0);

DIAMOND OPERATOR

Write:

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

Instead of:

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

Diamond operator: <>

Especially useful for super lengthy code:

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

WILL THIS COMPILE?

```
public class WillThisCompile {
        HashMap<String, HashMap<String, String>> hm;
         WillThisCompile(){
                hm = new HashMap<>();
        }
}
```

GENERICS

- Write and use parameterized types
- Avoids problems with unexpected input types during runtime
- You'll find them in the Java standard libraries a lot, but less common necessary to use for mainstream applications

GENERIC CLASSES

Add formal type parameter to class

```
public class Example<T>{
        private T exampleField;
        public T getExampleField(){
                return exampleField;
        }
}
```

Specify type when calling the class

```
Example<Animal> ex = new Example<>();
```

FORMAL TYPE PARAMETER

They are declared between < and >, after that they can be used

Naming conventions:

E: element

K: map key

V: map value

N: number

T, U, V: generic data types

EXERCISE: CREATE A CLASS WITH GENERIC DATA TYPE

- Make a new class Bag that takes a generic type T
- Add a class Groceries
- In a main method, instantiate a bag and specify Groceries as the type
- Make a list of groceries and create a bag using these groceries
- Make a new method to unpack groceries that takes a bag as parameter and returns each item as an object of type Groceries

TYPE ERASURE

- The compiler replaces all the generic data types to Objects
- So there's only one copy of the class with the generic type
- The compiler adds the necessary casts in your code

GENERIC INTERFACES

- Interfaces can declare formal type parameters too

```
public interface GenericTypeBehavior<T>{
        void beGeneric(T t);
}

public class Example<T> implements GenericTypeBehavior<T>{
        public void beGeneric(T t){}
}
```

EXERCISE GENERIC INTERFACE

- Make a generic interface Foldable
- Add one (abstract) method fold that takes the generic type and return it folded
- Implement the generic interface to the bag from the previous exercise

GENERIC METHODS

- Possible to declare formal type parameter on method level as well
- Can be used on static and non-static (static most common)

Syntax:

- Generic type declared directly before the return type

Modifiers <generic type> return type nameMethod(input parameters)

Example: public static <T> int giveFive(){

ARE THESE VALID METHOD DECLARATIONS?

1. public static T void methodName(T t)
2. public static void methodName(T t, U u)
3. public static <T> T methodName(T t)
4. public static T methodName(T t)
5. public static T <T> methodName(T t)
6. public static <T> GenericClass<T> methodName(T t)
7. public static T GenericClass<T> methodName(T t)

LEGACY CODE

- Before Java 1.4 there were no generics
- Collections without generics are called raw collections
- When some of the code has raw collections, exceptions can occur
- Java warns with a compile warning about unchecked and unsafe operations

Exercise: combine generics and raw collections to produce a ClassCastException during runtime

BOUNDS – FOR COLLECTIONS

- Generics that are treated as objects don't have a lot of methods available
- Solution for this problems: restricting what types can be used as generic in a collection
- This is done with a bounded parameter type
- Wildcard generic type >> ?
- This wildcard is used for an unknown generic type

BOUNDS – THE ISSUE

Java protects the user from doing funky stuff with collections, try:

```
List<Integer> list = new ArrayList<>();
list.add(new Integer(2));
List<Object> list2 = list;
list2.add("integer 2");
```

BOUNDS

- Unbounded wildcards: ?
  - Represents any data type
- Upper-bounded wildcards: ? extends Type
  - Any class that extends this type (or the type itself) can be used as formal parameter type
- Lower-bounded wildcards: ? super Type
  - The type itself or a superclass of that type

EXERCISE

- Solve this issue with bounds:

```
List<Integer> list = new ArrayList<>();
list.add(new Integer(2));
List<Object> list2 = list;
```

Bonus: Write the worst (working) code you can think of using bounds and generics. Explain why it is working.

WILL IT COMPILE?

```
Class B {}
Class A extends B {}
Class C extends A {}
```

1. `List<? extends B> list = new ArrayList<A>();`
2. `List<? extends B> list = new ArrayList<C>();`
3. `List<?> list = new ArrayList<C>();`
4. `List<? super B> list = new ArrayList<B>();`
5. `List<? super A> list = new ArrayList<C>();`
6. `List<? super A> list = new ArrayList<Object>();`
7. `List<?> list = new ArrayList<? extends B>();`

WILL IT COMPILE? - METHODS

```
<U> U methodName(List<U extends T> list {
        …correct body
}

<T> <? super T> methodName(List<? super T> list) {
        …correct body
}
```

LISTS, SETS, MAPS, QUEUES

- Collection is a group of objects in one object
- Collection framework exists of four interfaces:
  - List: ordered collection of elements, duplicates allowed, int index for accessing
  - Set: collection that doesn't allow duplicates
  - Queue: collection that orders elements in a specific order for processing, usually fifo
  - Map: maps key to values, no duplicate keys allowed


Collection
Map
List
Queue
Set

COLLECTION METHODS

- add() -> insert new element (returns boolean)
- remove() -> removes matching value/index (returns boolean)
- isEmpty() -> returns true if there are no elements in a collection
- size() -> returns the number of element in a collection
- clear() -> discard all elements in collection
- contains() -> return true if a value is in the collection

LIST INTERFACE

- Ordered collection that can contain duplicates
- Common implementations:
  - ArrayList -> resizable array, adding and removing slower than accessing
  - LinkedList -> implements list and queue interface, list with additional methods to add to beginning/end. Queue actions on it are fast, look up in the rest is slow
  - Vector -> old implementation for ArrayList that is slower, but it is thread safe
  - Stack -> old implementation of ArrayDeque, add and remove from the stack

CREATING A LIST WITH A FACTORY

- Arrays.asList(varargs) – fixed size list backed by array
- List.of(varargs) – immutable list
- List.copyOf(collection) – immutable list with copy of original collection values

EXERCISE: ARRAYLIST AND LINKEDLIST

- Make a new ArrayList, add 10 items to it
- Make a new LinkedList, add 10 items to it
- Make an array lookup in which you store 4 integers
- Look up the value of arraylist and linkedlist for all the indices in the array lookup

- Measure for ArrayList and LinkedList how long this takes on average running it 5 times
- Scale up by times 100 three times and see the difference. What is striking?
- Bonus: also measure the adding

BIGGEST DIFFERENCE

- Adding to the beginning:
  - Linkedlist, only needs to add a link
  - Arraylist needs to shift all the elements one position to the right

- Performance arraylist might be less terrible than expected for adding, because the reserved array is quite bit bigger than the used array, so no need to create a whole new array every single time

SET INTERFACE

- Doesn’t allow duplicate entries
- Common implementations:
  - HashSet -> stores element in hash tables, adding and finding element are fast, but the order in which they are added is not stored
  - TreeSet -> sorted order set, adding and finding element is slow, it implements the NavigableSet interface

EXERCISE: TREESET AND HASHSET

- Make a new TreeSet, add 10 items to it
- Make a new HashSet, add 10 the same items to it
- Make an array(list) lookup in which you store 4 integers
- Look up the value of lookup in TreeSet and HashSet

- Measure for TreeSet and HashSet how long this takes on average running it 5 times
- Scale up by times 100 three times and see the difference. What is striking?
- Bonus: also measure the adding

QUEUE INTERFACE

- Elements are added and removed in a specific order, usually FIFO
- Queue implementations:
  - LinkedList -> double ended queue that's also a List, not as efficient as a regular queue
  - ArrayDeque -> double ended queue, more effecient than LinkedList

QUEUE METHODS (IN ADDITION TO COLLECTION INTERFACE METHODS)

- Add -> adds element to back of the queue and return true or throws exception
- Element -> returns next element or throws exception when queue is empty
- Offer -> adds element to back of queue and return true for success and false for no success
- Remove -> removes and returns next element, or throws exception when queue is empty
- Push -> specific for double ended, adds element to front (stack, not on queue interface)
- Poll -> removes and returns next element, or returns null when empty
- Peek -> returns next element or null when queue is empty
- Pop -> removes and returns next element or throws exception when queue is empty (stack, not on queue interface)

ARRAYDEQUE METHODS (IN ADDITION TO COLLECTION INTERFACE METHODS)

Method description	Exception	No exception
Adds an element to back of the queue	add	Offer
Removes and returns next element	Remove (queue), pop (stack)	poll
Returns next element	element	Peek
Add element to front of the queue		push

EXERCISE: ARRAYDEQUE, STACK OR QUEUE?

- Write a program for having lunch with the whole group using ArrayDeque
- Everyone needs to stand in line to get lunch, add everyone to the queue and log the output of everyone getting lunch
- After lunch everyone piles their plates in the kitchen, log every stacking action
- Every day someone will do the dishes, log whose dish is being washed and put into the the drying rack
- Everyday someone will dry the plates and make a new pile in the cupboard and log the order

MAP INTERFACE

- Identify values by key, Map doesn't extend Collection interface
- Map implementations:
  - HashMap -> stores keys in hash table. Adding and finding have constant time, but order is not stored
  - TreeMap -> ordering storage by keys, adding and finding is slow
  - Hashtable (not HashTable) -> old implementation, old slow thread safe version of HashMap

MAP METHODS

- Put: adds or replaces key/value pair
- Get: returns the value based on the key parameter
- isEmpty: returns whether map is empty
- Size: returns number of key/value pairs in map
- Clear: remove all keys and values
- Remove: removes value based on key parameter
- containsKey: checks if map contains a certain key
- containsValue: checks if map contains a certain value
- keySet: returns a Set of all keys
- Values: returns a Collection of all values

OVERVIEW LIST, SET, MAP, QUEUE

Interface	Duplicates?	Ordered?	Key/value pairs?	Add/remove in specific order?
List	Yes	Yes	No	No
Set	No	No	No	No
Queue	Yes	Yes	No	Yes
Map	Values yes, keys no	No	Yes	No

EXERCISE: WHEN TO USE?

- Come up (individually) with a good example for:
  - Group A: ArrayList, TreeMap, LinkedList, TreeSet
  - Group B: ArrayDeque, HashSet, Stack, HashMap

COMPARATOR VS COMPARABLE

- Comparable:
  - Functional interface with method compareTo. Implementing this to your class makes the object comparable.
  - compareTo has three rules:
    - Should return 0 when it's compared to an equal element
    - Less than zero when it's compared to a bigger element
    - More than zero when it's compared to a smaller element
- Comparator:
  - Sort objects that don't implement comparable or make different ways of sorting for one object
  - Functional interface with method compare
  - Same rules as comparable

COMPARATOR VS COMPARABLE

- Comparable is used to mark an object as something that can be compared
- Comparator is used to implement a specific way of comparing an object
- Comparator is usually implemented with a lambda

COMPARATOR

```
Comparator<Person> byAge = (p1, p2)
-> p1.getAge()-p2.getAge();
```

COMPARABLE

```
Class Person implements Comparable<Person>{
        private int age;
        …
        public int compareTo(Person p){
                return this.age-p.getAge();
        }
}
```

EXERCISE

- Write a class Cat.
- Make it possible to compare two cats and order them by name (alphabetically)
- Add a way of ordering them by date of birth
- Add a way of ordering them by color light to dark

SEARCHING AND SORTING

- Collections.sort(collectionName) can be used for sorting if the class implements Comparable
- If it doesn't implement Comparable there is an option to send a specific comparator with it: Collections.sort(collectionName, comparatorName)
- The method binarySearch assumes the collection is sorted, but is not checking this. Using it on a not sorted collection gives unpredictable results

- Exercise: search for a cat, and sort the cats (log to console)

METHOD REFERENCES ::

- Can be used on:
  - Static methods
    - Consumer<List<Integer>> lam = l -> Collections.sort(l);
    - Consumer<List<Integer>> ref = Collections::sort;
  - Instance methods on a particular instance
    - String str = “Hello”;
    - Supplier<Boolean> lam = ()->str.isEmpty();
    - Supplier<Boolean> ref = str::isEmpty;
  - Instance methods on an instance determined during runtime
    - Predicate<String> lam = s->s.isEmpty();
    - Predicate<String> ref = String::isEmpty;
  - Constructors
    - Supplier<ArrayList> lam = () -> new ArrayList();
    - Supplier<ArrayList> ref = ArrayList::new;
- Java understands the exact parameters from the context

REMOVING CONDITIONALLY

Boolean removeIf(Predicate<? super E> filter)

Can be used to remove items from a collection based on a certain condition

Exercise: remove cats from your collection based on a certain condition (e.g. names starting with a C or the color being white)

UPDATING ALL ELEMENTS IN A LIST

void replaceAll(UnaryOperator<E> o)

Used for replacing all elements with another value

The interface UnaryOperator takes a parameter of a certain type and returns a parameter of the same type

Example: listName.replaceAll(x -> x+1)

Exercise: replace all cats with a subclass RobotCat

LOOPING THROUGH COLLECTION WITH LAMBDA

collectionName.forEach(Consumer c)

The interface Consumer takes a single parameter and doesn't return anything

Exercise: print the names of all cats

MAP API

- merge()
  - mapName.merge(obj1, obj2, biFunctionImpl);
  - BiFunction interface takes two parameters and returns a value
- computeIfPresent() -> not on OCP
  - BiFunction<T, U, V> mapper = some implementation
  - mapName.computeIfPresent(value, mapper)
  - Removes key if result of mapper is null
- computeIfAbsent() -> not on OCP
  - Function<T, U> mapper = some implementation
  - mapName.computeIfAbsent(key, mapper)

EXERCISE

- Create a new map to store first names and favorite art form.
- Use merge to add values of art form for the first names
- Use ComputeIfPresent to add ice skating as default for names starting with an M or S or J
- Use ComputeIfAbsent to add a new pair
- Use ComputeIfAbsent on an existing pair

CASE