JAVA Cheat Sheet 4
Data Structures and Advanced Concepts

Data Structures

Arrays

- Objects that store multiple variables of the same type.
- Can hold either primitives or object references.
- Will always be an object on the heap.

Declaration

```
// common
int[] key;
// better avoid due to readability
int key [];
```

Construction

Creates the object on the heap.

```
// The size has to be specified
int[] myArray = new int[3];
```

For multidimensional arrays omitting the second dimension's size is possible:

```
int[][] myArray = new int[3][];
```

Initializing

Arrays can be initialized by assigning values to their elements:

```
int[] myArray = int[3];
myArray[0] = 5;
```

The index always starts with 0 and the highest index equals the array's length -1.
Declaration, construction and initialization is also possible in one single step:

```
int[] dots = {1,3,5}
```

Anonymous Array Creation

It is possible to create an anonymous array, for example to pass it as a functions parameter.

```
functionCall(new int[]{1,2,3});
```

Enumerations

The JAVA type for enumerations is called enum.
The restrict the values for a variable to a predefined set of values.

```
enum CoffeeSize{
  BIG, HUGE, OVERWHELMING
};
```

The ';' at the end is optional.
An enum can be seen as a special kind of class. The values are constants (public static final) instances of this class. One can define constructors and other methods:

```
enum CoffeeSize{
  BIG(8), HUGE(10), OVERWHELMING(16);
  CoffeeSize(int ounces){
    this.ounces = ounces;
  }
  private int ounces;
  private int getOunces(){
    return this.ounces;
  }
}
```

Enums also can have a constant specific class body which overrides a method:

```
  ...
  OVERWHELMING(16){
    public String getLidCode(){
      return ”B”;
    }
  };
  ...
```

The enum-class implements equals and hashCode, therefore it is possible to use enums as keys for maps.

Lists, Trees, Collections

In JAVA we distinguish between **collections**, a **Collection** and the utility class **Collections**.
There are many classes considered to fulfil the concept of a collection. Those are collections. Then there is the class Collection, which is the interface for a subset of these collections. And then there is a utility class providing static methods for collections.
The single collections cannot take primitives as elements. In this case one have to use the wrapper types. Java implicitly uses **auto-boxing**, so it is possible to give primitives as arguments instead of their wrapper type representations.

Overview over the collections

- Collection
  - Set
    - HashSet
      - LinkedHashSet
    - SortedSet
  - List
    - ArrayList
    - Vector
    - LinkedList
  - Queue
    - LinkedList
    - PriorityQueue
- Map
  - HashTable
  - HashMap
    - LinkedHashMap
  - SortedMap
    - Navigable Map
      - TreeMap

Lists

There are three implementations of the interface List. They all provide the common methods of Collection and additionally provide methods related to the index of an element.

ArrayList
An ArrayList provides fast access and fast iterations over its elements.
But every time its allocated memory is exceeded by inserting new elements a new list has to be created and the whole old list has to copied into the new list.

```
List<String> list = new ArrayList<String>();
list.add(”hello”);
String string = list.get(0);
```

Vector
Leftover from earlier JAVA versions. Mainly the same as an ArrayList, but uses synchronized methods for the use in multi-threading, which makes it slower.

```
List<String> list = new Vector<String>();
list.add(”hello”);
String string = list.get(0);
```

LinkedList
A double-linked list. Provides methods for adding from the beginning or the end, which makes it a good choice for queues or stacks. Insertion and Deletion is faster than for the other two lists, but iteration and access may be slower.

```
List<String> list = new LinkedList<String>();
list.add(”hello”);
String string = list.get(0);
```

Trees

There are only two structures using trees:

- TreeSet
- TreeMap

Both provide an ascending order for the elements based on the elements' natural order.
The methods **lower**(<), **floor**(≤) and **higher**(>), **ceiling**(≥) can be used to navigate or search within tree structures.

Collections

This class contains utility methods for collections, for example to search elements, sort collections, reverse the order of a list and so on.

Advanced Concepts

Iterators

An **iterator** is an object that lets one loop over a collection step by step.

```
Iterator<Dog> iterator = d.iterator();
```

```
while(iterator.hasNext()){
  Dog dog = iterator.next();
  System.out.println(dog.name);
}
```

Recursion

Templates

Generics

Reflection

Annotation