Lecture 12
Collections and Generics

Quiz Time!

qui.su/Znva

Recap: recursion

- Functions calling themselves
- Divide & Conquer
  - **Divide** problem into pieces
  - **Conquer** pieces individually
  - **&**ombine partial solutions
- Has >=1 base case and >=1 recursive case

Recursion Recap: Counting Zeroes

Remember the Big Mean?

- We had a file full of numbers
- And we wanted to know the mean
- So we put the # of numbers at the top of the file
- But what if we don’t control the input?

(no, not this kind of mean)

Lists

The interface `List<E>` has the following methods:

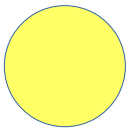
- `boolean add(E e)` (optional) – adds `e` to the end of the list
- `boolean contains(Object o)` – true if the list contains `o`
- `E get(int index)` – returns the element at position `index`
- `int indexOf(Object o)` – returns the index of `o` if `o` is in the list, otherwise -1
- `int size()` - number of elements in the list
- …

The classes that implement `List<E>` all implement these methods in different ways.

https://docs.oracle.com/javase/8/docs/api/java/util/List.html

Use interfaces for the types of variables:

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



List Example: Reversing Words

Lists

- Lists are handy when:
  - We don’t know in advance how many elements we will need
  - We want to add and remove elements later (not just overwrite old ones)
- They are not so good when:
  - We need very high performance
  - We need to minimize memory usage

Lists

- Different types of list implement the `List<T>` interface
- `LinkedList<T>`
  - Fast append, slow indexing
- `ArrayList<T>`
  - Fast indexing, slow append
- Use the right one for your use case

Boxing

- Primitive types (`int`, `boolean`, etc.) can not be used as type parameters
- Each primitive type has an equivalent reference type
  - `int` → `Integer`, `boolean` → `Boolean`, etc.
  - Primitive types are **unboxed**, reference equivalents are **boxed** and are called **wrapper classes** (“omslagsklasser”)
  - `Integer i = 42; // this is called` **`autoboxing`**
  - Boxed types are slower, take up more memory, but can be used as type parameters!
  - Always use equals for boxed types!

```
jshell> (Integer)42 == (Integer)42
$1 ==> true

jshell> (Integer)9001 == (Integer)9001
$2 ==> false
```

List Example:
Mean Computing

Maps (“avbildningstabeller”)

The interface `Map<K, V>` has the following methods:

- `boolean put(K key, V value)` (optional) – associates value with key
- `boolean containsKey(Object o)` – true if the map contains an entry with the key o
- `E get(K key)` – returns the element associated with the given key
- `int size()` - number of mappings in the map
- ...

https://docs.oracle.com/javase/8/docs/api/java/util/Map.html

Recap: `equals`

- Used to compare objects: `a.equals(b)`
- Follows a special pattern:

```
public class Pony {
  ...
  @Override
  public boolean equals(Object o) {
    if(this == o) {
      return true;
    }
    if(o == null || this.getClass() != o.getClass()){
      return false;
    }
    Pony other = (Pony)o;
    return this.name.equals(other.name) &&
           this.age == other.age &&
           ...
}
```

`equals` has a friend: `hashCode`

- Generates a **hash** of the object
- If `a.equals(b)`, then

  `a.hashCode() == b.hashCode()`

  - Does **NOT** apply the other way around!
    (E.g. `a.hashCode() == b.hashCode()` does not imply `a.equals(b)`)
- Used to speed up comparisons
  - ```
    if(a.hashCode() == b.hashCode()) {
      return a.equals(b);
    } else {
      return false;
    }
    ```
    ```

`equals` has a friend: `hashCode`

- Simplest valid implementation of equals:

```
@Override
public int hashCode() {
  return 0;
}
```

- A more useful implementation:

```
@Override
public int hashCode() {
  int hash = this.name.hashCode();
  hash = hash*97 + this.skill.hashCode();
  hash = hash*97 + this.age;
  ...
  return hash;
}
```

HashMap

- Efficient implementation of `Map<K, V>` based on hashing
- Unordered, but useful in most circumstances when you want a map
- Performance and correctness depends on `K.hashCode()` being correct and well-written

Map Example:
Ponies with Driving Licenses

Map Example:
Word Frequencies

Generic Methods

A method signature may have type parameters:

```
public static <T> List<T> replicate(int copies, T elem) {
  List<T> list = new ArrayList<T>();
  for(int i = 0; i < copies; i++) {
    list.add(elem);
  }
  return list;
}
```

Inside the method, we may use type parameters like any other type.

- Variables can have type S, T or S[] or List<S> or …

However, we cannot write new S();

Generic Classes

Classes may also have type parameters:

```
public class Pair<S, T> {
    …
}
```

Now we have classes

- `Pair<Integer, Integer>`
- `Pair<String, Double>`
- `Pair<Pair<Integer, Integer>, Double>`
- etc.

A type parameter must be instantiated with a class (not a primitive data type).

Abstract classes and interfaces may have type parameters.

Reading and Exercises

- Reading
  - 3.7, 3.10, 10.12.3, 17.1.1, 17.1.2, 17.2, 17.3, 17.7
- Exercises
  - 17.9, exercise 5
  - Bonus exercise: Contact List (see course website)