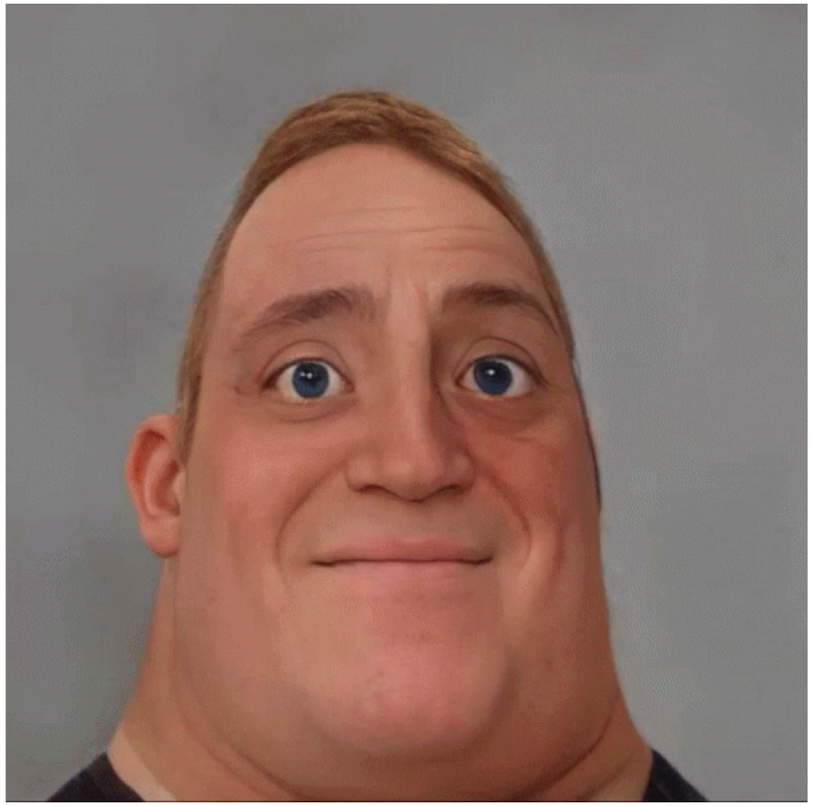
Welcome to CS2030 Lab 5!

11 March 2022 [10J]

Please login to plab now!!

🤒 Heavy content ahead!



Practical Assessment

- Deadline: Sunday 2359
- Must get A! Otherwise moderation might be harsh... Watch recitation for guide

Project

- The project has been released on CodeCrunch
- The entire project is worth 10% of your grade
- Same as previous labs, submit your files onto CodeCrunch!
- **Style and design is graded!** You will need to write Javadocs for public methods
- More levels to come :)

Mini-Lecture I:

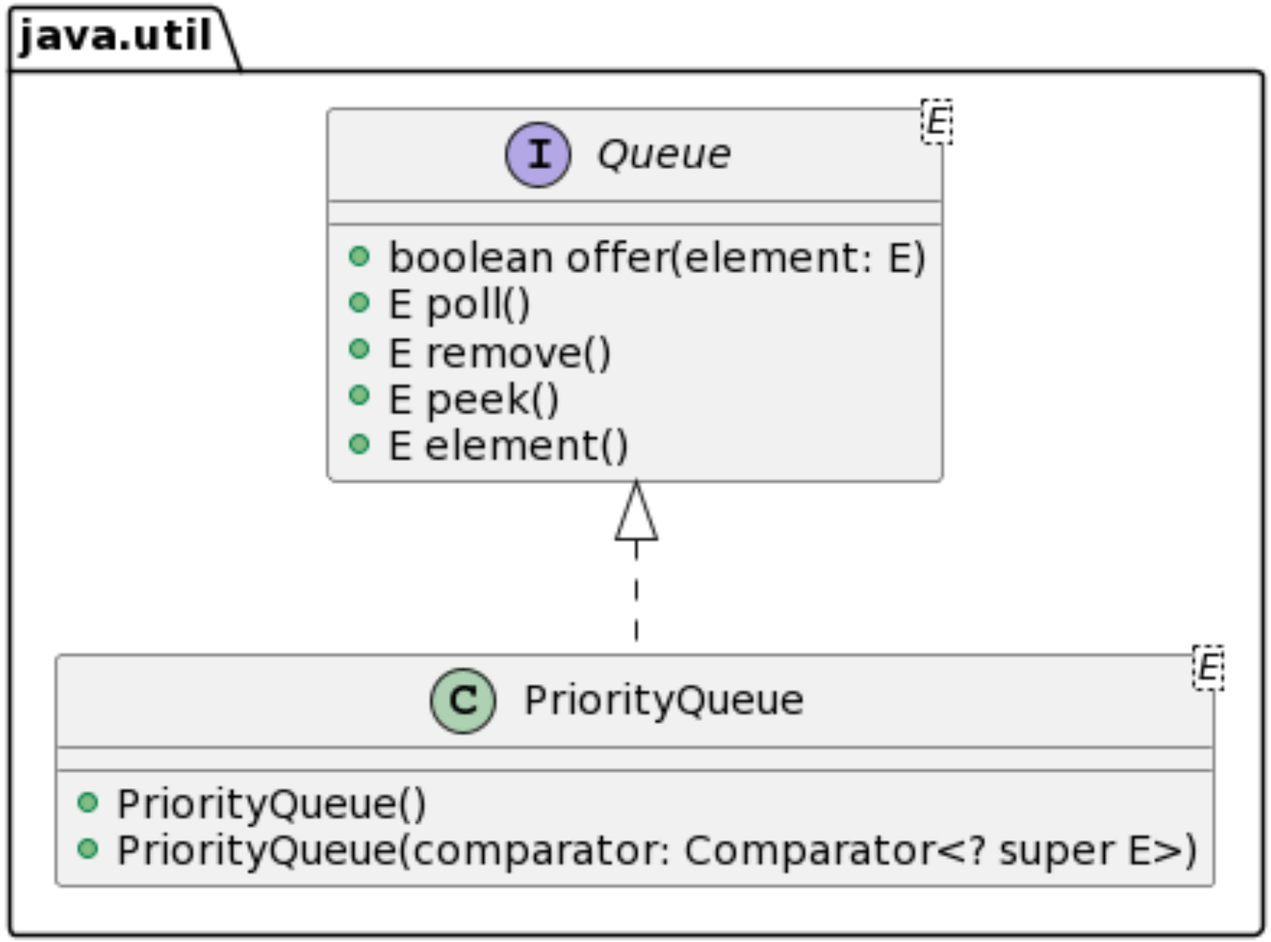
Priority Queues, `Comparable` and `Comparator` interfaces

Priority Queues

In a priority queue (PQ), the element with the **highest priority*** is removed first.

- A **queue** is a first-in, first-out data structure.
- Elements are appended to the end of the queue and removed from the head of the queue.
- In a PQ, elements are assigned priorities. When accessing the elements, the element with the highest priority is removed first.

`java.util.PriorityQueue<E>`



- `offer`: inserts an element into the queue.
- `poll`: Retrieves and removes the head of this queue, or `null` if this queue is empty.
- `remove`: Retrieves and removes the head of this queue, or throws an exception if this queue is empty.
- `peek`: Retrieves, but does not remove, the head of this queue, returning `null` if this queue is empty.
- `element`: Retrieves, but does not remove, the head of this queue, throws an exception if this queue is empty.

How to define priority?

The `Comparable<T>` Interface

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

Many classes in the Java library such as `Integer`, `Double` and `String` already implement it to define a natural order.

```
jshell> new Integer(3).compareTo(new Integer(5));
$1 ===> -1
jshell> "ABC".compareTo("ABC"); // lexicographical order
$2 ===> 0
jshell> new Double(2.1).compareTo(new Double(2));
$3 ===> 1
```

```java
// Compare ages, then names if ages are the same
class Person implements Comparable<Person> {
    int age;
    String name;

    // Other code here

    @Override
    public int compareTo(Person other) {
        if (this.age != other.age) {
            return this.age - other.age;
        }
        return this.name.compareTo(other.name);
    }
}
```

In the above example, we compare names if the ages are the same

java.util.Comparator<T>

`Comparator` can be used to compare the objects of a class that **doesn't implement** `Comparable` or **define a new criteria for comparing objects**.

```
import java.util.Comparator;

class AgeComparator implements Comparator<Person> {
    // Sort by age in ascending order
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```java
import java.util.PriorityQueue;
import java.util.Comparator;

class PriorityQueueDemo {
    public static void main(String[] args) {
        PriorityQueue<String> queue1 = new PriorityQueue<>();
        queue1.offer("Oklahoma");
        queue1.offer("Indiana");
        queue1.offer("Georgia");
        queue1.offer("Texas");
        System.out.println("PQ using Comparable:");
        while (!queue1.isEmpty()) {
            System.out.print(queue1.remove() + " ");
        }

        PriorityQueue<String> queue2 = new PriorityQueue<>(Comparator.reverseOrder());
        queue2.offer("Oklahoma");
        queue2.offer("Indiana");
        queue2.offer("Georgia");
        queue2.offer("Texas");
        System.out.println("\nPQ using Comparator:");
        while (!queue2.isEmpty()) {
            System.out.print(queue2.remove() + " ");
        }
    }
}
```

```
PQ using Comparable:
Georgia Indiana Oklahoma Texas
PQ using Comparator:
Texas Oklahoma Indiana Georgia
```

Mini Lecture II: Generics

define:generic

relating to or characteristic of a whole group or class
// "Romantic comedy" is the *generic* term for such films.

- You have just seen generic class `PriorityQueue` and generic interfaces `Comparable` and `Comparator`!
- *Generics* let you parameterise types - define a class or method with generic types that the compiler can replace with concrete types.
- e.g.: From the generic class `PriorityQueue`, you can create `PriorityQueue` objects for holding strings or numbers. Here strings and numbers are concrete types that replace the generic type.

- A generic class or method permits you to specify allowable types of objects that the class or method can work with. **If you attempt to use an incompatible object, the compiler will detect that error**.
- By convention, a single capital letter such as E or T is used to denote a formal generic type.
- Generic types must be reference types. You cannot replace a generic type with a primitive type such as `int`, `double`, or `char`. For example the following statement is wrong

```
ArrayList<int> intList new ArrayList<>();
```

Defining Generic Classes and Interfaces

A generic type can be defined for a class or interface. A concrete type must be specified when using the class to create an object or using the class or interface to declare a reference variable.

- Occasionally, a generic class may have more than one parameter. In this case, place the parameters together inside the brackets, separated by commas—for example, `<E1, E2, E3>`.

Generic Methods

A generic type can be defined for a static method.

```
public class GenericMethodDemo {
    public static void main(String[] args) {
        Integer[] integers = {1, 2, 3, 4, 5};
        String[] stirngs = {"London", "Paris", "New York", "Austin"};

        // invoking generic method
        GenericMethodDemo.<Integer>print(integers);
        GenericMethodDemo.<String>print(strings);
    }

    public static <E> void print(E[] list) { // declaring generic method
        for (int i = 0; i < list.length; i++)
            System.out.print(list[i] + " ");
        System.out.println();
    }
}
```

Generic Methods

A generic type can be defined for a static method.

```
public class GenericMethodDemo {
    public static void main(String[] args) {
        Integer[] integers = {1, 2, 3, 4, 5};
        String[] stirngs = {"London", "Paris", "New York", "Austin"};

        // also works, compiler automatically discovers the actual type.
        print(integers);
        print(strings);
    }

    public static <E> void print(E[] list) { // declaring generic method
        for (int i = 0; i < list.length; i++)
            System.out.print(list[i] + " ");
        System.out.println();
    }
}
```

Bounds for Type Variables

We want to compute the smallest element of an array. **What's wrong?**

```
public static <T> T min(T[] a) {
    if (a == null || a.length == 0)
        return null;
    T smallest = a[0];
    for (int i = 1; i < a.length; i++)
        if (smallest.compareTo(a[i]) > 0)
            smallest = a[i];
    return smallest
}
```

How do we know that the class to which T belongs has a `compareTo` method?

Solution

```
public static <T extends Comparable<T>> T min(T[] a)...
```

1. T is a subtype of `Comparable`.

2. The elements to be compared are of the T type.

Java Generics

```
public class Box<T> {
    private T item;
    public Box(T item) {
        this.item = item;
    }
}
```

T is a generic type.

Declaring `Box<Integer>` replaces all instances of `T` in your code with `Integer`.

Java Generics

- Every instance of `Box` can have a different type that is assigned to T.
- Therefore, T belongs to an **instance** of `Box`.
- How do we insert a generic type into a static method or variable?

Java Generics

```
public class Box<T> {
    private T item;
    public static <T> Box<T> empty() {
        // Pink T replaces all the Ts in this method
    }
}
```

Solution: Declare <T> in front of a static method.

Java Generics

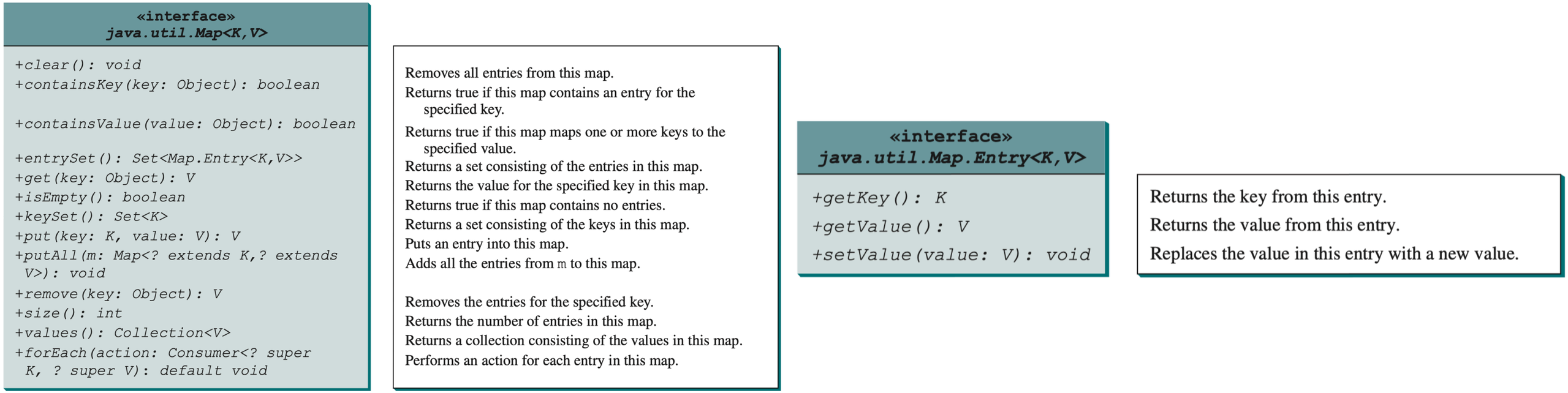
⚠️ The T's on the right side **only exist within the scope of the static method**, while the T's on the left exist in all other **instance attributes** of `Box`.

```
public class Box<T> {
    private T item;
    public static <T> Box<T> empty() {
        // code
    }
}
```

```
public class Box<T> {
    private T item;
    public static <T> Box<T> empty() {
        // code
    }
}
```

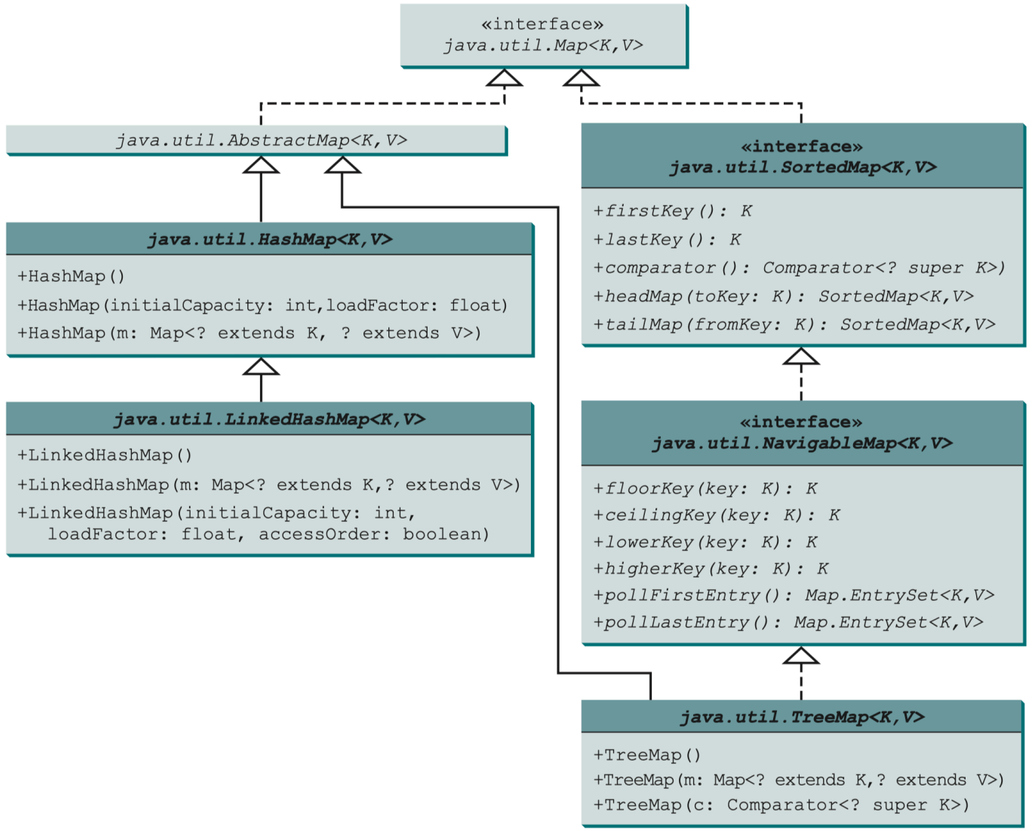
`java.util.Map<K, V>`

- A map is a data structure which maps a key to a value.
- One key can only be mapped to one value.
- Mapping a an existing key to another value will replace the current mapping in the map.
- Think of it as a dictionary (for students familiar with Python/ JavaScript).
- Frequently referred to as **Hash Table**! (or Associative Array)


«interface»
java.util.Map<K,V>
+clear(): void
+containsKey(key: Object): boolean
+containsValue(value: Object): boolean
+entrySet(): Set<Map.Entry<K,V>>
+get(key: Object): V
+isEmpty(): boolean
+keySet(): Set<K>
+put(key: K, value: V): V
+putAll(m: Map<? extends K,? extends V>): void
+remove(key: Object): V
+size(): int
+values(): Collection<V>
+forEach(action: Consumer<? super K, ? super V): default void
Removes all entries from this map.
Returns true if this map contains an entry for the specified key.
Returns true if this map maps one or more keys to the specified value.
Returns a set consisting of the entries in this map.
Returns the value for the specified key in this map.
Returns true if this map contains no entries.
Returns a set consisting of the keys in this map.
Puts an entry into this map.
Adds all the entries from m to this map.
Removes the entries for the specified key.
Returns the number of entries in this map.
Returns a collection consisting of the values in this map.
Performs an action for each entry in this map.
«interface»
java.util.Map.Entry<K,V>
+getKey(): K
+getValue(): V
+setValue(value: V): void
Returns the key from this entry.
Returns the value from this entry.
Replaces the value in this entry with a new value.

Under the hood

💡If you don’t need to maintain an order in a map when updating it, use a `HashMap`. When you need to maintain the insertion order or access order in the map, use a `LinkedHashMap`. When you need the map to be sorted on keys, use a `TreeMap`.



java.util.LinkedHashMap<K, V>

- `LinkedHashMap` is a `Map` that is implemented by `HashMap` and `LinkedList` (you will learn more about them in CS2040/C/S)
- Require two generic types
  - `K` - key
  - `V` - value
- To declare a `LinkedHashMap` with keys of type `A` and values of type `B`:

```
Map<A, B> map = new LinkedHashMap<A, B>();
```

Maps are implemented using hashing.

Hashing is a technique that retrieves the value using the index obtained from the key without performing a search.

The `hashCode` method

- A hash code is an integer that is derived from an object. (some kind of identity tag for the machine to read)
  - If `x` and `y` are two distinct objects, there should be a high probability that `x.hashCode()` and `y.hashCode()` are different.
- If you redefine the `equals` method, you will also need to redefine the `hashCode` method for objects that users might insert into a hash table.
- **The `hashCode` method should return an integer (which can be negative)**. Just combine the hash codes of the instance fields so that the hash codes for different objects are likely to be widely scattered.

```java
class A {
    private final int a;
    private final String b;

    @Override
    public int hashCode() {
        return a.hashCode() + b.hashCode();
    }
}
```

Simpler solution:

```java
import java.util.Objects;
// modify hashCode method to become:
public int hashCode() {
    return Objects.hash(a, b);
}
```

- Your definitions of `equals` and `hashCode` must be compatible: If `x.equals(y)` is `true`, then `x.hashCode()` must return the same value as `y.hashCode()`.

If you want to use a custom class as a **key** in a `LinkedHashMap`, you **must** override the equals and `hashCode` methods, which are inherited from the `Object` class.

Extra: Java being Java

```
jshell> List<Integer> list = List.of(127, 128)
list ⟹ [127, 128]

jshell> Integer num1 = 127
num1 ⟹ 127

jshell> Integer num2 = 128
num2 ⟹ 128

jshell> num1 == list.get(0)
$4 ⟹ true

jshell> num2 == list.get(1) // ??
$5 ⟹ false

jshell> int num3 = num2
num3 ⟹ 128

jshell> num3 == list.get(1)
$7 ⟹ true

jshell> num2.equals(list.get(1))
$8 ⟹ true
```

Moral of the story:

Always use `.equals` when comparing reference types!

If you are curious about this phenomenon, read up on *Integer cache*.