合肥學院
HEFEI UNIVERSITY

OOP with Java

22. Collections, `equals`, and `hashCode`

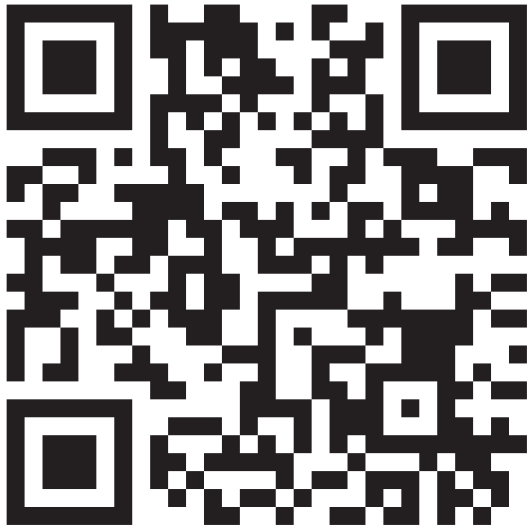
Thomas Weise · 汤卫思

tweise@hfuu.edu.cn · http://iao.hfuu.edu.cn

Hefei University, South Campus 2
Faculty of Computer Science and Technology
Institute of Applied Optimization
230601 Shushan District, Hefei, Anhui, China
Econ. & Tech. Devel. Zone, Jinxiu Dadao 99

合肥学院 南艳湖校区/南2区
计算机科学与技术系
应用优化研究所
中国 安徽省 合肥市 蜀山区 230601
经济技术开发区 锦绣大道99号

Outline



website

Introduction

- Java provides a lot of utility classes

Introduction

- Java provides a lot of utility classes
- The most important ones are probably the collection classes

Introduction

- Java provides a lot of utility classes
- The most important ones are probably the collection classes
- Collections are objects which can store other objects

Introduction

- Java provides a lot of utility classes
- The most important ones are probably the collection classes
- Collections are objects which can store other objects
- The most important collection types are lists, maps, and sets

Introduction

- Java provides a lot of utility classes
- The most important ones are probably the collection classes
- Collections are objects which can store other objects
- The most important collection types are lists, maps, and sets
- You can find their default implementations in package `java.util`

Lists

- Arrays in Java have a fixed length

Lists

- Arrays in Java have a fixed length
- Lists are a dynamic-length version of arrays

Lists

- Arrays in Java have a fixed length
- Lists are a dynamic-length version of arrays
- They offer a lot of advanced functionality

Lists

- Arrays in Java have a fixed length
- Lists are a dynamic-length version of arrays
- They offer a lot of advanced functionality: you can
  - get the object at list index `i` via `get(i)`

Lists

- Arrays in Java have a fixed length
- Lists are a dynamic-length version of arrays
- They offer a lot of advanced functionality: you can
  - get the object at list index `i` via `get(i)`
  - get store an object `o` at list index `i` via `set(i, o)`

Lists

- Arrays in Java have a fixed length
- Lists are a dynamic-length version of arrays
- They offer a lot of advanced functionality: you can
  - get the object at list index `i` via `get(i)`
  - get store an object `o` at list index `i` via `set(i, o)`
  - add an object `o` via `add(o)`

Lists

- Arrays in Java have a fixed length
- Lists are a dynamic-length version of arrays
- They offer a lot of advanced functionality: you can
  - get the object at list index `i` via `get(i)`
  - get store an object `o` at list index `i` via `set(i, o)`
  - add an object `o` via `add(o)`
  - insert an object `o` at index `i` via `add(i, o)`

Lists

- Arrays in Java have a fixed length
- Lists are a dynamic-length version of arrays
- They offer a lot of advanced functionality: you can
  - get the object at list index `i` via `get(i)`
  - get store an object `o` at list index `i` via `set(i, o)`
  - add an object `o` via `add(o)`
  - insert an object `o` at index `i` via `add(i, o)`
  - deletes then object at index `i` via `remove(i)`

Lists

- Arrays in Java have a fixed length
- Lists are a dynamic-length version of arrays
- They offer a lot of advanced functionality: you can
  - get the object at list index `i` via `get(i)`
  - get store an object `o` at list index `i` via `set(i, o)`
  - add an object `o` via `add(o)`
  - insert an object `o` at index `i` via `add(i, o)`
  - deletes then object at index `i` via `remove(i)`
  - add/remove all objects in another collection `c` via `addAll(c)` / `removeAll(c)`

Lists

- Arrays in Java have a fixed length
- Lists are a dynamic-length version of arrays
- They offer a lot of advanced functionality: you can
  - get the object at list index `i` via `get(i)`
  - get store an object `o` at list index `i` via `set(i, o)`
  - add an object `o` via `add(o)`
  - insert an object `o` at index `i` via `add(i, o)`
  - deletes then object at index `i` via `remove(i)`
  - add/remove all objects in another collection `c` via `addAll(c)` / `removeAll(c)`
  - iterate over a list `l` in the same read-only fashion as used for arrays via `for(Object e : l){ ... }`

Lists

- Arrays in Java have a fixed length
- Lists are a dynamic-length version of arrays
- They offer a lot of advanced functionality: you can
  - get the object at list index `i` via `get(i)`
  - get store an object `o` at list index `i` via `set(i, o)`
  - add an object `o` via `add(o)`
  - insert an object `o` at index `i` via `add(i, o)`
  - deletes then object at index `i` via `remove(i)`
  - add/remove all objects in another collection `c` via `addAll(c)` / `removeAll(c)`
  - iterate over a list `l` in the same read-only fashion as used for arrays via `for(Object e : l){ ... }`
  - ...

List Implementations

- Java provides several different implementations of this functionality

List Implementations

- Java provides several different implementations of this functionality
- `java.util.ArrayList` is the implementation we always use

List Implementations

- Java provides several different implementations of this functionality
- `java.util.ArrayList` is the implementation we always use
- `java.util.Vector` is basically the same, just slower (due to synchronization, which is useless anyway)

List Implementations

- Java provides several different implementations of this functionality
- `java.util.ArrayList` is the implementation we always use
- `java.util.Vector` is basically the same, just slower (due to synchronization, which is useless anyway)
- `java.util.LinkedList` is another slower implementation of the same functionality (yes, someone will say linked lists are efficient for some special cases, blablabla, but even if you have millions of elements, `LinkedList` will just consome more memory and be slower than `ArrayList`)

Example for using `ArrayList`

Listing: Example for using `ArrayList`

```
import java.util.ArrayList;

/** a test for array list, which allows us to store and manipulate a sequence of objects */
public class ArrayListTest {
  /** The main routine
   *  @param args  we ignore this parameter */
  public static void main(String[] args) {
    ArrayList<String> list = new ArrayList<>();

    list.add("Hello"); //$NON-NLS-1$
    list.add("World."); //$NON-NLS-1$
    list.add("It's"); //$NON-NLS-1$
    list.add("me."); //$NON-NLS-1$
    System.out.println(list); // [Hello, World., It's, me.]

    int index = list.indexOf("World."); //$NON-NLS-1$
    System.out.println(index); // 1

    list.remove(index);
    System.out.println(list); // [Hello, It's, me.]

    list.add(index, "World!!!"); //$NON-NLS-1$
    System.out.println(list); // [Hello, World!!!, It's, me]

    for(String string : list) { // fast read-only iteration
      System.out.print(string); // HelloWorld!!!It'sme.
    }
    System.out.println();

    ArrayList<String> list2 = new ArrayList<>();
    list2.addAll(list);
    list2.remove(1);
    System.out.println(list2); // [Hello, It's, me.]

    list.removeAll(list2);
    System.out.println(list); // [World!!!]

    list.addAll(list2);
    list.addAll(list);
    System.out.println(list); // [World!!!, Hello, It's, me., World!!!, Hello, It's, me.]
  }
}
```

Object Equality and Identity

- As we know, comparing object variables/expression with `==` only yields `true` if both sides reference the exact same object, i.e., on identity, i.e., if they point to the same memory location

Object Equality and Identity

- As we know, comparing object variables/expression with `==` only yields `true` if both sides reference the exact same object, i.e., on identity, i.e., if they point to the same memory location
- Often, you want to compare based on object content, not just by reference, i.e., based on equality

Object Equality and Identity

- As we know, comparing object variables/expression with `==` only yields `true` if both sides reference the exact same object, i.e., on identity, i.e., if they point to the same memory location
- Often, you want to compare based on object content, not just by reference, i.e., based on equality
- If you implement a class holding an integer, you want two instances to be considered as equal if they have the same integer value stored in them

Object Equality and Identity

- As we know, comparing object variables/expression with `==` only yields `true` if both sides reference the exact same object, i.e., on identity, i.e., if they point to the same memory location
- Often, you want to compare based on object content, not just by reference, i.e., based on equality
- If you implement a class holding an integer, you want two instances to be considered as equal if they have the same integer value stored in them
- Class `Object` provides a method `public boolean equals(Object)` intended for this purpose

Object Equality and Identity

- As we know, comparing object variables/expression with `==` only yields `true` if both sides reference the exact same object, i.e., on identity, i.e., if they point to the same memory location
- Often, you want to compare based on object content, not just by reference, i.e., based on equality
- If you implement a class holding an integer, you want two instances to be considered as equal if they have the same integer value stored in them
- Class `Object` provides a method `public boolean equals(Object)` intended for this purpose
- Any subclass can override it to perform a class-specific comparison for equality

Object Equality and Identity

- As we know, comparing object variables/expression with `==` only yields `true` if both sides reference the exact same object, i.e., on identity, i.e., if they point to the same memory location
- Often, you want to compare based on object content, not just by reference, i.e., based on equality
- If you implement a class holding an integer, you want two instances to be considered as equal if they have the same integer value stored in them
- Class `Object` provides a method `public boolean equals(Object)` intended for this purpose
- Any subclass can override it to perform a class-specific comparison for equality
- By default, it just does the same as `==` if you do not override it

Object Equality and Identity

- As we know, comparing object variables/expression with `==` only yields `true` if both sides reference the exact same object, i.e., on identity, i.e., if they point to the same memory location
- Often, you want to compare based on object content, not just by reference, i.e., based on equality
- If you implement a class holding an integer, you want two instances to be considered as equal if they have the same integer value stored in them
- Class `Object` provides a method `public boolean equals(Object)` intended for this purpose
- Any subclass can override it to perform a class-specific comparison for equality
- By default, it just does the same as `==` if you do not override it
- Java collections use `equals` instead of `==` to compare objects when you search inside them

Object Equality and Identity

- As we know, comparing object variables/expression with `==` only yields `true` if both sides reference the exact same object, i.e., on identity, i.e., if they point to the same memory location
- Often, you want to compare based on object content, not just by reference, i.e., based on equality
- If you implement a class holding an integer, you want two instances to be considered as equal if they have the same integer value stored in them
- Class `Object` provides a method `public boolean equals(Object)` intended for this purpose
- Any subclass can override it to perform a class-specific comparison for equality
- By default, it just does the same as `==` if you do not override it
- Java collections use `equals` instead of `==` to compare objects when you search inside them
- `equals` must be implemented in a way so that `a.equals(b)== b.equals(a)` !

integer holder class without `equals` override

Listing: integer holder class without `equals` override

```java
/** a holder class for an integer */
public final class IntHolder {
  /** the integer value */
  private final int value;

  /** create the integer value holder */
  public IntHolder(final int _value) {
    this.value = _value;
  }

  /** get the string representation of this value */
  @Override
  public String toString() {
    return "" + this.value; //$NON-NLS-1$
  }

  /** The main routine
   *  @param args  we ignore this parameter */
  public static void main(String[] args) {
    IntHolder a = new IntHolder(1);
    IntHolder b = new IntHolder(2);
    IntHolder c = new IntHolder(1);

    System.out.print(a == a); System.out.print('␣'); System.out.println(a.equals(a)); // true  true
    System.out.print(a == b); System.out.print('␣'); System.out.println(a.equals(b)); // false false
    System.out.print(a == c); System.out.print('␣'); System.out.println(a.equals(c)); // false false
    System.out.print(b == a); System.out.print('␣'); System.out.println(b.equals(a)); // false false
    System.out.print(b == b); System.out.print('␣'); System.out.println(b.equals(b)); // true  true
    System.out.print(b == c); System.out.print('␣'); System.out.println(b.equals(c)); // false false
    System.out.print(c == a); System.out.print('␣'); System.out.println(c.equals(a)); // false false
    System.out.print(c == b); System.out.print('␣'); System.out.println(c.equals(b)); // false false
    System.out.print(c == c); System.out.print('␣'); System.out.println(c.equals(c)); // true  true
  }
}
```

integer holder class with `equals` override

Listing: integer holder class with equals override

```java
/** a holder class for an integer */
public final class IntHolderWithEquals {
  /** the integer value */
  private final int value;

  /** create the integer value holder */
  public IntHolderWithEquals(final int _value) {
    this.value = _value;
  }

  /** get the string representation of this value */
  @Override
  public String toString() {
    return "" + this.value; //$NON-NLS-1$
  }

  /** override the equals method from Object checking for equality */
  @Override
  public boolean equals(final Object o) {
    return ((o instanceof IntHolderWithEquals) && // check if right class
            (((IntHolderWithEquals)o).value == this.value));
  }


  /** The main routine
   *  @param args  we ignore this parameter */
  public static void main(String[] args) {
    IntHolderWithEquals a = new IntHolderWithEquals(1);
    IntHolderWithEquals b = new IntHolderWithEquals(2);
    IntHolderWithEquals c = new IntHolderWithEquals(1);

    System.out.print(a == a); System.out.print('␣'); System.out.println(a.equals(a)); // true  true
    System.out.print(a == b); System.out.print('␣'); System.out.println(a.equals(b)); // false false
    System.out.print(a == c); System.out.print('␣'); System.out.println(a.equals(c)); // false true
    System.out.print(b == a); System.out.print('␣'); System.out.println(b.equals(a)); // false false
    System.out.print(b == b); System.out.print('␣'); System.out.println(b.equals(b)); // true  true
    System.out.print(b == c); System.out.print('␣'); System.out.println(b.equals(c)); // false false
    System.out.print(c == a); System.out.print('␣'); System.out.println(c.equals(a)); // false true
    System.out.print(c == b); System.out.print('␣'); System.out.println(c.equals(b)); // false false
    System.out.print(c == c); System.out.print('␣'); System.out.println(c.equals(c)); // true  true
  }
}
```

integer holder without `equals` override in list

Listing: integer holder without equals override in list

```
import java.util.ArrayList;

/** a test for search methods in array list */
public class ArrayListWithoutEqualsTest {
  /** The main routine
   *  @param args  we ignore this parameter */
  public static void main(String[] args) {
    ArrayList<IntHolder> list = new ArrayList<>();

    list.add(new IntHolder(3));
    IntHolder ih4 = new IntHolder(4);
    list.add(ih4);
    list.add(new IntHolder(-1));
    list.add(new IntHolder(3));
    System.out.println(list); // [3, 4, -1, 3]

    System.out.println(list.contains(new IntHolder(3)));     // false
    System.out.println(list.contains(new IntHolder(4)));     // false
    System.out.println(list.contains(ih4));                  // true
    System.out.println(list.contains(new IntHolder(5)));     // false

    System.out.println(list.indexOf(new IntHolder(3)));      // -1 (not found)
    System.out.println(list.indexOf(new IntHolder(4)));      // -1 (not found)
    System.out.println(list.indexOf(ih4));                   // 1  (found)
    System.out.println(list.indexOf(new IntHolder(5)));      // -1 (not found)

    System.out.println(list.lastIndexOf(new IntHolder(3))); // -1 (not found)
    System.out.println(list.lastIndexOf(new IntHolder(4))); // -1 (not found)
    System.out.println(list.lastIndexOf(ih4));               // 1  (found)
    System.out.println(list.lastIndexOf(new IntHolder(5))); // -1 (not found)
  }
}
```

integer holder with `equals` override in list

Listing: integer holder with equals override in list

```
import java.util.ArrayList;

/** a test for search methods in array list */
public class ArrayListWithEqualsTest {
  /** The main routine
   *  @param args  we ignore this parameter */
  public static void main(String[] args) {
    ArrayList<IntHolderWithEquals> list = new ArrayList<>();

    list.add(new IntHolderWithEquals(3));
    IntHolderWithEquals ih4 = new IntHolderWithEquals(4);
    list.add(ih4);
    list.add(new IntHolderWithEquals(-1));
    list.add(new IntHolderWithEquals(3));
    System.out.println(list); // [3, 4, -1, 3]

    System.out.println(list.contains(new IntHolderWithEquals(3)));    // true
    System.out.println(list.contains(new IntHolderWithEquals(4)));    // true
    System.out.println(list.contains(ih4));                           // true
    System.out.println(list.contains(new IntHolderWithEquals(5)));    // false

    System.out.println(list.indexOf(new IntHolderWithEquals(3)));     // 0
    System.out.println(list.indexOf(new IntHolderWithEquals(4)));     // 1
    System.out.println(list.indexOf(ih4));                            // 1  (found)
    System.out.println(list.indexOf(new IntHolderWithEquals(5)));     // -1 (not found)

    System.out.println(list.lastIndexOf(new IntHolderWithEquals(3))); // 3
    System.out.println(list.lastIndexOf(new IntHolderWithEquals(4))); // 1
    System.out.println(list.lastIndexOf(ih4));                        // 1  (found)
    System.out.println(list.lastIndexOf(new IntHolderWithEquals(5))); // -1 (not found)
  }
}
```

Maps

- In Lesson 18: *Visibility, Encapsulation,* `final`*, and Inner Classes*, we had an example for a data structure storing key-value relationships, i.e., a map

Maps

- In Lesson 18: *Visibility, Encapsulation,* `final`*, and Inner Classes*, we had an example for a data structure storing key-value relationships, i.e., a map
- Java provides utility classes for this purpose, which can store, for each (unique) key, one associated value

Maps

- In Lesson 18: *Visibility, Encapsulation,* `final`*, and Inner Classes*, we had an example for a data structure storing key-value relationships, i.e., a map
- Java provides utility classes for this purpose, which can store, for each (unique) key, one associated value
- There are quite a few implementations of that functionality

Maps

- In Lesson 18: *Visibility, Encapsulation,* `final` *, and Inner Classes*, we had an example for a data structure storing key-value relationships, i.e., a map
- Java provides utility classes for this purpose, which can store, for each (unique) key, one associated value
- There are quite a few implementations of that functionality:
  - `java.util.HashMap`: This is the implementation we will always use

Maps

- In Lesson 18: *Visibility, Encapsulation,* `final`*, and Inner Classes*, we had an example for a data structure storing key-value relationships, i.e., a map
- Java provides utility classes for this purpose, which can store, for each (unique) key, one associated value
- There are quite a few implementations of that functionality:
  - `java.util.HashMap`: This is the implementation we will always use
  - `java.util.HashTable`: A slower implementation of the same functionality (due to useless synchronization)

Maps

- In Lesson 18: *Visibility, Encapsulation,* `final`*, and Inner Classes*, we had an example for a data structure storing key-value relationships, i.e., a map
- Java provides utility classes for this purpose, which can store, for each (unique) key, one associated value
- There are quite a few implementations of that functionality:
  - `java.util.HashMap`: This is the implementation we will always use
  - `java.util.HashTable`: A slower implementation of the same functionality (due to useless synchronization)
  - `java.util.Dictionary`: An obselete implementation of similar functionality (never use this one)

Example for using `HashMap`

Listing: Example for using `HashMap`

```
import java.util.HashMap;

/** a test for Hash Map, which relates key objects to value objects, each key must be unique */
public class HashMapTest {
  /** The main routine
   *  @param args  we ignore this parameter */
  public static void main(String[] args) {
    HashMap<String,String> map = new HashMap<>();

    map.put("Hello",      "World!"); //$NON-NLS-1$ //$NON-NLS-2$
    map.put("It's",       "me!");    //$NON-NLS-1$ //$NON-NLS-2$
    map.put("Professor", "Weise");   //$NON-NLS-1$ //$NON-NLS-2$
    map.put("Weise",      "Thomas"); //$NON-NLS-1$ //$NON-NLS-2$
    map.put("Teacher",    "Weise"); //$NON-NLS-1$ //$NON-NLS-2$
    System.out.println(map); // {Professor=Weise, Hello=World!, Weise=Thomas, It's=me!, Teacher=Weise}

    map.put("It's",       "you, not me!"); //$NON-NLS-1$ //$NON-NLS-2$
    map.put("Professor", "Jacky Chan");    //$NON-NLS-1$ //$NON-NLS-2$
    System.out.println(map); // {Professor=Jacky Chan, Hello=World!, Weise=Thomas, It's=you, not me!, Teacher=Weise}

    System.out.println(map.remove("Professor")); // Jacky Chan //$NON-NLS-1$
    System.out.println(map); // {Hello=World!, Weise=Thomas, It's=you, not me!, Teacher=Weise}

    System.out.println(map.entrySet()); // [Hello=World!, Weise=Thomas, It's=you, not me!, Teacher=Weise]
    System.out.println(map.keySet());   // [Hello, Weise, It's, Teacher]
    System.out.println(map.values());   // [World!, Thomas, you, not me!, Weise]

    HashMap<String,String> other = new HashMap<>();
    other.put("Hello",   "China"); //$NON-NLS-1$ //$NON-NLS-2$
    other.put("Country", "China"); //$NON-NLS-1$ //$NON-NLS-2$
    other.put("Weise",   "Teacher"); //$NON-NLS-1$ //$NON-NLS-2$
    System.out.println(other); // {Hello=China, Weise=Teacher, Country=China}

    map.putAll(other);
    System.out.println(map);  // {Hello=China, Weise=Teacher, It's=you, not me!, Teacher=Weise, Country=China}

    for(String key : map.keySet()) {
      System.out.println(map.get(key)); // China \n Teacher \n you, not me! \n Weise \n China
    }
  }
}
```

Maps with our own Key Classes

- Sometimes, you may want to use your class as key type for a map

Maps with our own Key Classes

- Sometimes, you may want to use your class as key type for a map
- This is dangerous

Maps with our own Key Classes

- Sometimes, you may want to use your class as key type for a map
- This is dangerous
- How does a map work?

Maps with our own Key Classes

- Sometimes, you may want to use your class as key type for a map
- This is dangerous
- How does a map work?
  - A `HashMap` in Java tries to provide very fast access

Maps with our own Key Classes

- Sometimes, you may want to use your class as key type for a map
- This is dangerous
- How does a map work?
  - A `HashMap` in Java tries to provide very fast access
  - Therefore, it internally uses an array `table` of entries

Maps with our own Key Classes

- Sometimes, you may want to use your class as key type for a map
- This is dangerous
- How does a map work?
  - A `HashMap` in Java tries to provide very fast access
  - Therefore, it internally uses an array `table` of entries
  - When looking up a key, it first converts it into an `int`

Maps with our own Key Classes

- Sometimes, you may want to use your class as key type for a map
- This is dangerous
- How does a map work?
  - A `HashMap` in Java tries to provide very fast access
  - Therefore, it internally uses an array `table` of entries
  - When looking up a key, it first converts it into an `int`
  - Then wraps this `int` into the range `0` ... `table.length-1`

Maps with our own Key Classes

- Sometimes, you may want to use your class as key type for a map
- This is dangerous
- How does a map work?
  - A `HashMap` in Java tries to provide very fast access
  - Therefore, it internally uses an array `table` of entries
  - When looking up a key, it first converts it into an `int`
  - Then wraps this `int` into the range `0` ... `table.length-1`
  - This is where a linked list of entries with the keys mapping to the same index is located

Maps with our own Key Classes

- Sometimes, you may want to use your class as key type for a map
- This is dangerous
- How does a map work?
  - A `HashMap` in Java tries to provide very fast access
  - Therefore, it internally uses an array `table` of entries
  - When looking up a key, it first converts it into an `int`
  - Then wraps this `int` into the range `0` ... `table.length-1`
  - This is where a linked list of entries with the keys mapping to the same index is located
  - Ideally, the list is either only 1 element long or `null`, so it is immediately clear whether the key is in the map or not

Maps with our own Key Classes

- Sometimes, you may want to use your class as key type for a map
- This is dangerous
- How does a map work?
  - A `HashMap` in Java tries to provide very fast access
  - Therefore, it internally uses an array `table` of entries
  - When looking up a key, it first converts it into an `int`
  - Then wraps this `int` into the range `0` ... `table.length-1`
  - This is where a linked list of entries with the keys mapping to the same index is located
  - Ideally, the list is either only 1 element long or `null`, so it is immediately clear whether the key is in the map or not
  - Otherwise, we can find it somewhere in the list (comparing keys via `equals`)

Maps with our own Key Classes

- Sometimes, you may want to use your class as key type for a map
- This is dangerous
- How does a map work?
  - A `HashMap` in Java tries to provide very fast access
  - Therefore, it internally uses an array `table` of entries
  - When looking up a key, it first converts it into an `int`
  - Then wraps this `int` into the range `0` ... `table.length-1`
  - This is where a linked list of entries with the keys mapping to the same index is located
  - Ideally, the list is either only 1 element long or `null`, so it is immediately clear whether the key is in the map or not
  - Otherwise, we can find it somewhere in the list (comparing keys via `equals`)
  - If there are too many elements in the map compared to `table.length`, the table is resized

Maps with our own Key Classes

- Sometimes, you may want to use your class as key type for a map
- This is dangerous
- How does a map work?
- What does this mean?

Maps with our own Key Classes

- Sometimes, you may want to use your class as key type for a map
- This is dangerous
- How does a map work?
- What does this mean?
  - Essentially, we need to provide a way for the map to translate our key objects to `int` s

Maps with our own Key Classes

- Sometimes, you may want to use your class as key type for a map
- This is dangerous
- How does a map work?
- What does this mean?
  - Essentially, we need to provide a way for the map to translate our key objects to `int` s
  - This is done via the `public int hashCode()` method of class `Object`

Maps with our own Key Classes

- Sometimes, you may want to use your class as key type for a map
- This is dangerous
- How does a map work?
- What does this mean?
  - Essentially, we need to provide a way for the map to translate our key objects to `int` s
  - This is done via the `public int hashCode()` method of class `Object`
  - Which does, by default, return something like the memory address of the object

Maps with our own Key Classes

- Sometimes, you may want to use your class as key type for a map
- This is dangerous
- How does a map work?
- What does this mean?
  - Essentially, we need to provide a way for the map to translate our key objects to `int` s
  - This is done via the `public int hashCode()` method of class `Object`
  - Which does, by default, return something like the memory address of the object
- So what does this mean?

Maps with our own Key Classes

- Sometimes, you may want to use your class as key type for a map
- This is dangerous
- How does a map work?
- What does this mean?
  - Essentially, we need to provide a way for the map to translate our key objects to `int` s
  - This is done via the `public int hashCode()` method of class `Object`
  - Which does, by default, return something like the memory address of the object
- So what does this mean?
- If we do not override `public int hashCode()` to use our object's data instead of the memory address, `HashMap` will (almost always) be unable to find our keys...

Maps with our own Key Classes

- Sometimes, you may want to use your class as key type for a map
- This is dangerous
- How does a map work?
- What does this mean?
  - Essentially, we need to provide a way for the map to translate our key objects to `int` s
  - This is done via the `public int hashCode()` method of class `Object`
  - Which does, by default, return something like the memory address of the object
- So what does this mean?
- If we do not override `public int hashCode()` to use our object's data instead of the memory address, `HashMap` will (almost always) be unable to find our keys...
- Of course we also need to override `equals` !

integer holder class without `hashCode` override

Listing: integer holder class without `hashCode` override

```java
/** a holder class for an integer */
public final class IntHolderWithEqualsWithoutHashCode {
  /** the integer value */
  private final int value;

  /** create the integer value holder */
  public IntHolderWithEqualsWithoutHashCode(final int _value) {
    this.value = _value;
  }

  /** get the string representation of this value */
  @Override
  public String toString() {
    return "" + this.value; //$NON-NLS-1$
  }

  /** override the equals method from Object checking for equality */
  @Override
  public boolean equals(final Object o) {
    return ((o instanceof IntHolderWithEqualsWithoutHashCode) && // check if right class
            (((IntHolderWithEqualsWithoutHashCode)o).value == this.value));
  }


  /** The main routine
   *  @param args  we ignore this parameter */
  public static void main(String[] args) {
    IntHolderWithEqualsWithoutHashCode a = new IntHolderWithEqualsWithoutHashCode(1);
    IntHolderWithEqualsWithoutHashCode b = new IntHolderWithEqualsWithoutHashCode(2);
    IntHolderWithEqualsWithoutHashCode c = new IntHolderWithEqualsWithoutHashCode(1);

    System.out.println(a.hashCode());  // this will print
    System.out.println(b.hashCode());  // three entirely different
    System.out.println(c.hashCode());  // numbers

    System.out.print(a == a); System.out.print('␣'); System.out.println(a.equals(a)); // true  true
    System.out.print(a == b); System.out.print('␣'); System.out.println(a.equals(b)); // false false
    System.out.print(a == c); System.out.print('␣'); System.out.println(a.equals(c)); // false true
    System.out.print(b == a); System.out.print('␣'); System.out.println(b.equals(a)); // false false
    System.out.print(b == b); System.out.print('␣'); System.out.println(b.equals(b)); // true  true
    System.out.print(b == c); System.out.print('␣'); System.out.println(b.equals(c)); // false false
    System.out.print(c == a); System.out.print('␣'); System.out.println(c.equals(a)); // false true
    System.out.print(c == b); System.out.print('␣'); System.out.println(c.equals(b)); // false false
    System.out.print(c == c); System.out.print('␣'); System.out.println(c.equals(c)); // true  true
  }
}
```

Listing: integer holder without hashCode override in hash map

```
import java.util.HashMap;

/** a test for Hash Map */
public class HashMapWithoutHashCodeTest {
  /** The main routine
   *  @param args  we ignore this parameter */
  public static void main(String[] args) {
    HashMap<IntHolderWithEqualsWithoutHashCode, String> map = new HashMap<>();

    map.put(new IntHolderWithEqualsWithoutHashCode(1), "A"); //$NON-NLS-1$
    System.out.println(map); // {1=A}
    map.put(new IntHolderWithEqualsWithoutHashCode(2), "B"); //$NON-NLS-1$
    System.out.println(map); // {1=A, 2=B}
    map.put(new IntHolderWithEqualsWithoutHashCode(3), "C"); //$NON-NLS-1$
    System.out.println(map); // {1=A, 2=B, 3=C}
    map.put(new IntHolderWithEqualsWithoutHashCode(1), "D"); //$NON-NLS-1$
    System.out.println(map); // {1=A, 2=B, 1=D, 3=C}
    map.put(new IntHolderWithEqualsWithoutHashCode(3), "E"); //$NON-NLS-1$
    System.out.println(map); // {1=A, 2=B, 1=D, 3=C, 3=E}

    System.out.println(map.get(new IntHolderWithEqualsWithoutHashCode(1))); // null <- key lost
    System.out.println(map.get(new IntHolderWithEqualsWithoutHashCode(2))); // null <- key lost
    System.out.println(map.get(new IntHolderWithEqualsWithoutHashCode(3))); // null <- key lost
    System.out.println(map.get(new IntHolderWithEqualsWithoutHashCode(4))); // null <- key does not
        exist
  }
}
```

integer holder class with `hashCode` override

Listing: integer holder class with `hashCode` override

```java
/** a holder class for an integer */
public final class IntHolderWithEqualsAndHashCode {
  /** the integer value */
  private final int value;

  /** create the integer value holder */
  public IntHolderWithEqualsAndHashCode(final int _value) {
    this.value = _value;
  }

  /** get the string representation of this value */
  @Override
  public String toString() {
    return "" + this.value; //$NON-NLS-1$
  }

  /** override the equals method from Object checking for equality */
  @Override
  public boolean equals(final Object o) {
    return ((o instanceof IntHolderWithEqualsAndHashCode) && // check if right class
            (((IntHolderWithEqualsAndHashCode)o).value == this.value));
  }

  /** override the hashCode method from Object to return an integer number representing this instance (luckily, our only instance variable is an int)*/
  @Override
  public int hashCode() { // normally, your class will have more complex member variables, say objects, strings, doubles, etc.
    return this.value;    // you would then return some combination of their hashCodes()
  }                       // the hash code of a list, for instance, is something like a special sum of its element's hash codes


  /** The main routine
   *  @param args  we ignore this parameter */
  public static void main(String[] args) {
    IntHolderWithEqualsAndHashCode a = new IntHolderWithEqualsAndHashCode(1);
    IntHolderWithEqualsAndHashCode b = new IntHolderWithEqualsAndHashCode(2);
    IntHolderWithEqualsAndHashCode c = new IntHolderWithEqualsAndHashCode(1);

    System.out.println(a.hashCode());  // 1 <-- this has changed, we now get the same
    System.out.println(b.hashCode());  // 2 <-- hash codes for the same data
    System.out.println(c.hashCode());  // 1 <-- see?

    System.out.print(a == a); System.out.print('␣'); System.out.println(a.equals(a)); // true  true
    System.out.print(a == b); System.out.print('␣'); System.out.println(a.equals(b)); // false false
    System.out.print(a == c); System.out.print('␣'); System.out.println(a.equals(c)); // false true
    System.out.print(b == a); System.out.print('␣'); System.out.println(b.equals(a)); // false false
    System.out.print(b == b); System.out.print('␣'); System.out.println(b.equals(b)); // true  true
    System.out.print(b == c); System.out.print('␣'); System.out.println(b.equals(c)); // false false
    System.out.print(c == a); System.out.print('␣'); System.out.println(c.equals(a)); // false true
    System.out.print(c == b); System.out.print('␣'); System.out.println(c.equals(b)); // false false
    System.out.print(c == c); System.out.print('␣'); System.out.println(c.equals(c)); // true  true
  }
}
```

integer holder with `hashCode` override in hash map

Listing: integer holder with `hashCode` override in hash map

```
import java.util.HashMap;

/** a test for Hash Map */
public class HashMapWithHashCodeTest {
  /** The main routine
   *  @param args  we ignore this parameter */
  public static void main(String[] args) {
    HashMap<IntHolderWithEqualsAndHashCode, String> map = new HashMap<>();

    map.put(new IntHolderWithEqualsAndHashCode(1), "A"); //$NON-NLS-1$
    System.out.println(map); // {1=A}
    map.put(new IntHolderWithEqualsAndHashCode(2), "B"); //$NON-NLS-1$
    System.out.println(map); // {1=A, 2=B}
    map.put(new IntHolderWithEqualsAndHashCode(3), "C"); //$NON-NLS-1$
    System.out.println(map); // {1=A, 2=B, 3=C}
    map.put(new IntHolderWithEqualsAndHashCode(1), "D"); //$NON-NLS-1$
    System.out.println(map); // {1=D, 2=B, 3=C}
    map.put(new IntHolderWithEqualsAndHashCode(3), "E"); //$NON-NLS-1$
    System.out.println(map); // {1=D, 2=B, 3=E}

    System.out.println(map.get(new IntHolderWithEqualsAndHashCode(1))); // D
    System.out.println(map.get(new IntHolderWithEqualsAndHashCode(2))); // B
    System.out.println(map.get(new IntHolderWithEqualsAndHashCode(3))); // E
    System.out.println(map.get(new IntHolderWithEqualsAndHashCode(4))); // null  <- key does not exist
  }
}
```

Relationship of `equals` and `hashCode`

- There is a very simple and important relationship between `equals` and `hashCode`

Relationship of `equals` and `hashCode`

- There is a very simple and important relationship between `equals` and `hashCode`:

if `a.equals(b)` then it must hold that `a.hashCode()== b.hashCode()`

Relationship of `equals` and `hashCode`

- There is a very simple and important relationship between `equals` and `hashCode`:

if `a.equals(b)` then it must hold that `a.hashCode()== b.hashCode()`

- This means that, whenever we override `equals`, we also need to override `hashCode` and vice versa

Relationship of `equals` and `hashCode`

- There is a very simple and important relationship between `equals` and `hashCode`:

if `a.equals(b)` then it must hold that `a.hashCode()== b.hashCode()`

- This means that, whenever we override `equals`, we also need to override `hashCode` and vice versa
- But remember, this is a one-way relationship

Relationship of `equals` and `hashCode`

- There is a very simple and important relationship between `equals` and `hashCode`:

if `a.equals(b)` then it must hold that `a.hashCode()== b.hashCode()`

- This means that, whenever we override `equals`, we also need to override `hashCode` and vice versa
- But remember, this is a one-way relationship
- If two objects have the same hash code, they do not necessarily need to be equal, i.e., from `a.hashCode()== b.hashCode()` it does not follow that `a.equals(b)`

Sets

- A `Set` is a data structure which can either contain or not contain an element

Sets

- A `Set` is a data structure which can either contain or not contain an element
- Different from lists, each element can occur at most once

Sets

- A `Set` is a data structure which can either contain or not contain an element
- Different from lists, each element can occur at most once
- You can imagine it as a map with object keys and `Boolean` values

Sets

- A `Set` is a data structure which can either contain or not contain an element
- Different from lists, each element can occur at most once
- You can imagine it as a map with object keys and `Boolean` values (actually, it is not that far from this in reality)
- Your keys for the set must implement both `equals` and `hashCode`

Sets

- A `Set` is a data structure which can either contain or not contain an element
- Different from lists, each element can occur at most once
- You can imagine it as a map with object keys and `Boolean` values (actually, it is not that far from this in reality)
- Your keys for the set must implement both `equals` and `hashCode`
- We will always use the Java utility class `java.util.HashSet` for representing sets

Example for using `HashSet`

Listing: Example for using HashSet

```
import java.util.HashSet;

/** a test for sets, a set can contain each element exactly once */
public class HashSetTest {
  /** The main routine
   *  @param args  we ignore this parameter */
  public static void main(String[] args) {
    HashSet<String> set = new HashSet<>();

    System.out.println(set.add("Hello"));    // true  //$NON-NLS-1$
    System.out.println(set);                 // [Hello]
    System.out.println(set.add("World!"));   // true  //$NON-NLS-1$
    System.out.println(set);                 // [Hello, World!]
    System.out.println(set.add("World!"));   // false //$NON-NLS-1$
    System.out.println(set);                 // [Hello, World!]
    System.out.println(set.add("It's"));     // true //$NON-NLS-1$
    System.out.println(set);                 // [Hello, World!, It's]
    System.out.println(set.add("me!"));      // true //$NON-NLS-1$
    System.out.println(set);                 // [Hello, World!, It's, me!]

    System.out.println(set.contains("It's"));// true  //$NON-NLS-1$
    System.out.println(set.remove("It's"));  // true  //$NON-NLS-1$
    System.out.println(set);                 // [Hello, World!, me!
    System.out.println(set.contains("It's"));// false //$NON-NLS-1$]
    System.out.println(set.remove("It's"));  // false //$NON-NLS-1$
    System.out.println(set);                 // [Hello, World!, me!]
  }
}
```

Summary

- We have learned about the basic collections offerent by Java
- These include Lists, Maps, and Sets
- Using them properly with our own classes requires us to override the methods `public boolean equals(Object)` and `public int hashCode()` inherited from class `Object`
- We must ensure that `a.equals(b)` $\Longrightarrow$ `a.hashCode()== b.hashCode()`
- We noticed that all of Java's collections make heavy use of generics we discussed in Lesson 21: *Generics*

谢谢

Thank you

Thomas Weise [汤卫思]
tweise@hfuu.edu.cn
http://iao.hfuu.edu.cn

Hefei University, South Campus 2
Institute of Applied Optimization
Shushan District, Hefei, Anhui,
China

Caspar David Friedrich, “Der Wanderer über dem Nebelmeer”, 1818
http://en.wikipedia.org/wiki/Wanderer_above_the_Sea_of_Fog