Java

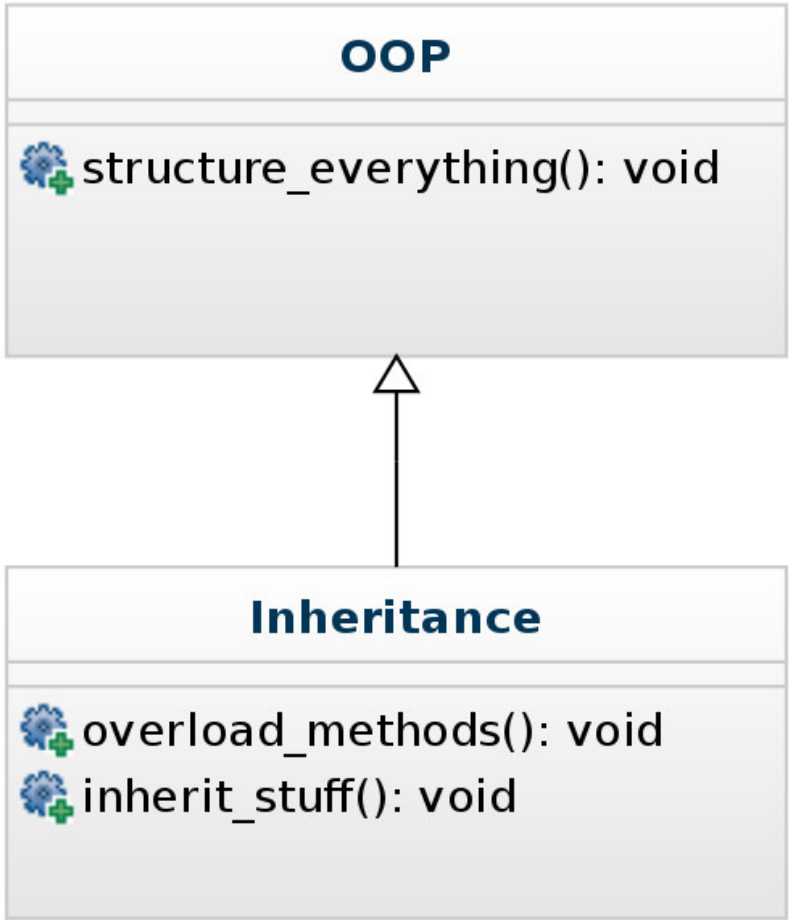
Generics and Collections

Nico Westerbeck

October 11, 2016

Overview

Inheritance



Calling static Methods

```
1       class EnrollmentSystem {
2           public static void doSomething() {
3           }
4       }
5
6       [...]
7       EnrollmentSystem example = new EnrollmentSystem();
8       example.doSomething(); // Works
9
10      EnrollmentSystem.doSomething();  // Works aswell
```

No fields available!

```
1       class EnrollmentSystem {
2           private int x;
3
4           public static void doSomething() {
5               x = 5;  // Does not work!
6           }
7       }
```

No fields available!

```
1       class EnrollmentSystem {
2           private static int x;
3
4           public static void doSomething() {
5               x = 5;  // Works
6           }
7       }
```

Avoid this unless you know what it does!

Generics

```
1        List<Tutor> list1 = new LinkedList<Tutor>();
2        List<Student> list2 = new LinkedList<Student>();
3
```

Generics

```
1  public class LinkedList<T> {
2      T[] items;
3      int nextFreeItem;
4
5      public void add(T newItem) {
6          items[nextFreeItem] = newItem;
7          nextFreeItem++;
8      }
9  }
```

Why they suck

- You can not use operators on them
- Compile-time-generics only
- Can't use builtin-types, needs wrapper classes (see next slides)
- Wildcarting: List¡?¿

=> I won't teach them in detail

Wrapper Class

Primitive data types can not be elements in collections. Use wrapper classes like *Integer* instead.

```
1       public static void main(String[] args) {
2
3           LinkedList<Integer> list = new LinkedList<Integer>()
        ;
4
5           list.add(3);
6           list.addFirst(1);
7           list.add(3);
8           list.add(8);
9           list.remove(3); // remove the 4th element
10          list.add(7);
11
12          System.out.println(list); // prints: [1, 3, 3, 7]
13      }
14
```

Wrapper Class

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

Wrapper Class

Wrapper classes hold extra functionality related to their datatype

```
1       public static void main(String[] args) {
2
3           Integer example = Integer.valueOf("12345");
4
5           System.out.println(example);  // Prints 12345
6
7           System.out.println(Integer.toBinaryString(example));
        // Prints 11000000111001
8
9       }
10
```

Collections Framework

Java offers various data structures like **Sets**, **Lists** and **Maps**. Those structures are part of the collections framework.

There are interfaces to access the data structures in an easy way. There are multiple implementations for various needs. Alternatively you can use your own implementations.

For more information visit:

- `https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html`
- `https://docs.oracle.com/javase/7/docs/api/java/util/Set.html`
- `https://docs.oracle.com/javase/7/docs/api/java/util/List.html`
- `https://docs.oracle.com/javase/7/docs/api/java/util/Map.html`

Set

A set is a collection that holds one type of objects. A set can not contain one element twice. Like all collections the interface *Set* is part of the package `java.util`.

```
1   import java.util.*;
2
3   public class TestSet {
4
5       public static void main(String[] args) {
6           Set<String> set = new HashSet<String>();
7
8           set.add("foo");
9           set.add("bar");
10          set.remove("foo");
11          System.out.println(set); // prints: [bar]
12      }
13  }
14
```

In the following examples `import java.util.*;` will be omitted.

List

A list is an ordered collection.

The implementation `LinkedList` is a double-linked list.

```
1       public static void main(String[] args) {
2
3           List<String> list = new LinkedList<String>();
4
5           list.add("foo");
6           list.add("foo"); // insert "foo" at the end
7           list.add("bar");
8           list.add("foo");
9           list.remove("foo"); // removes the first "foo"
10
11          System.out.println(list); // prints: [foo, bar, foo]
12      }
13
```

List Methods

some useful List methods:

void	`add(int index, E element)`	insert element at position index
E	`get(int index)`	get element at position index
E	`set(int index, E element)`	replace element at position index
E	`remove(int index)`	remove element at position index

some useful LinkedList methods:

void	`addFirst(E element)`	append element to the beginning
E	`getFirst()`	get first element
void	`addLast(E element)`	append element to the end
E	`getLast()`	get last element

For Loop

The for loop can iterate over every element of a collection:

`for (E e : collection)`

```
1       public static void main(String[] args) {
2
3           List<Integer> list =
4               new LinkedList<Integer>();
5
6           list.add(1);
7           list.add(3);
8           list.add(3);
9           list.add(7);
10
11          for (Integer i : list) {
12              System.out.print(i + " "); // prints: 1 3
    3 7
13          }
14      }
15
```

Iterator

An iterator iterates step by step over a collection.

```
1       public static void main(String[] args) {
2
3           List<Integer> list = new LinkedList<Integer>();
4
5           list.add(1);
6           list.add(3);
7           list.add(3);
8           list.add(7);
9
10          Iterator<Integer> iter = list.iterator();
11
12          while (iter.hasNext()) {
13              System.out.print(iter.next());
14          }
15          // prints: 1337
16      }
17
```

Iterator

A standard iterator has only three methods:

- `boolean hasNext()` - indicates if therer are more elements
- `E next()` - returns the next element
- `void remove()` - returns the current element

The iterator is instanced via `collection.iterator()` :

```
1 Collection<E> collection = new Implementation<E>;
2 Iterator<E> iter = collection.iterator();
3
```

Special iterators like *ListIterator* are more sophisticated.

Map

The interface *Map* is not a subinterface of *Collection*.
A map contains pairs of key and value. Each key refers to a value. Two keys can refer to the same value. There are not two equal keys in one map. *Map* is part of the package `java.util`.

```
1       public static void main (String[] args) {
2
3           Map<Integer, String> map =
4               new HashMap<Integer, String>();
5
6           map.put(23, "foo");
7           map.put(28, "foo");
8           map.put(31, "bar");
9           map.put(23, "bar"); // "bar" replaces "foo" for key
        = 23
10
11          System.out.println(map);
12          // prints: {23=bar, 28=foo, 31=bar}
13      }
14
```

KeySet and Values

You can get the set of keys from the map. Because one value can exist multiple times a collection is used for the values.

```
1       public static void main (String[] args) {
2
3           // [...] map like previous slide
4
5           Set<Integer> keys = map.keySet();
6           Collection<String> values = map.values();
7
8           System.out.println(keys);
9           // prints: [23, 28, 31]
10
11          System.out.println(values);
12          // prints: [bar, foo, bar]
13      }
14
```

Iterator

Therer is no iterator for maps. To iterate over a map use the iterator from the set of keys.

```
1       public static void main (String[] args) {
2
3           // [...] map, keys, values like previous slide
4           Iterator<Integer> iter = keys.iterator();
5
6           while(iter.hasNext()) {
7               System.out.print(map.get(iter.next()) + " ");
8           } // prints: bar foo bar
9
10          System.out.println(); // print a line break
11
12          for(Integer i: keys) {
13              System.out.print(map.get(i) + " ");
14          } // prints: bar foo bar
15      }
16
```

Nested Maps

Nested maps offer storage with key pairs.

```
1       public static void main (String[] args) {
2
3           Map<String, Map<Integer, String>> addresses =
4               new HashMap<String, Map<Integer, String>>();
5
6           addresses.put("Noethnitzer Str.",
7               new HashMap<Integer, String>());
8
9           addresses.get("Noethnitzer Str.").
10              put(46, "Andreas-Pfitzmann-Bau");
11          addresses.get("Noethnitzer Str.").
12              put(44, "Fraunhofer IWU");
13      }
14
```

Overview

List	Keeps order of objects Easily traversible Search not effective
Set	No duplicates No order - still traversible Effective searching
Map	Key-Value storage "Search" super-effective Traversing difficult