Java for Advanced Programmers

Java Collections

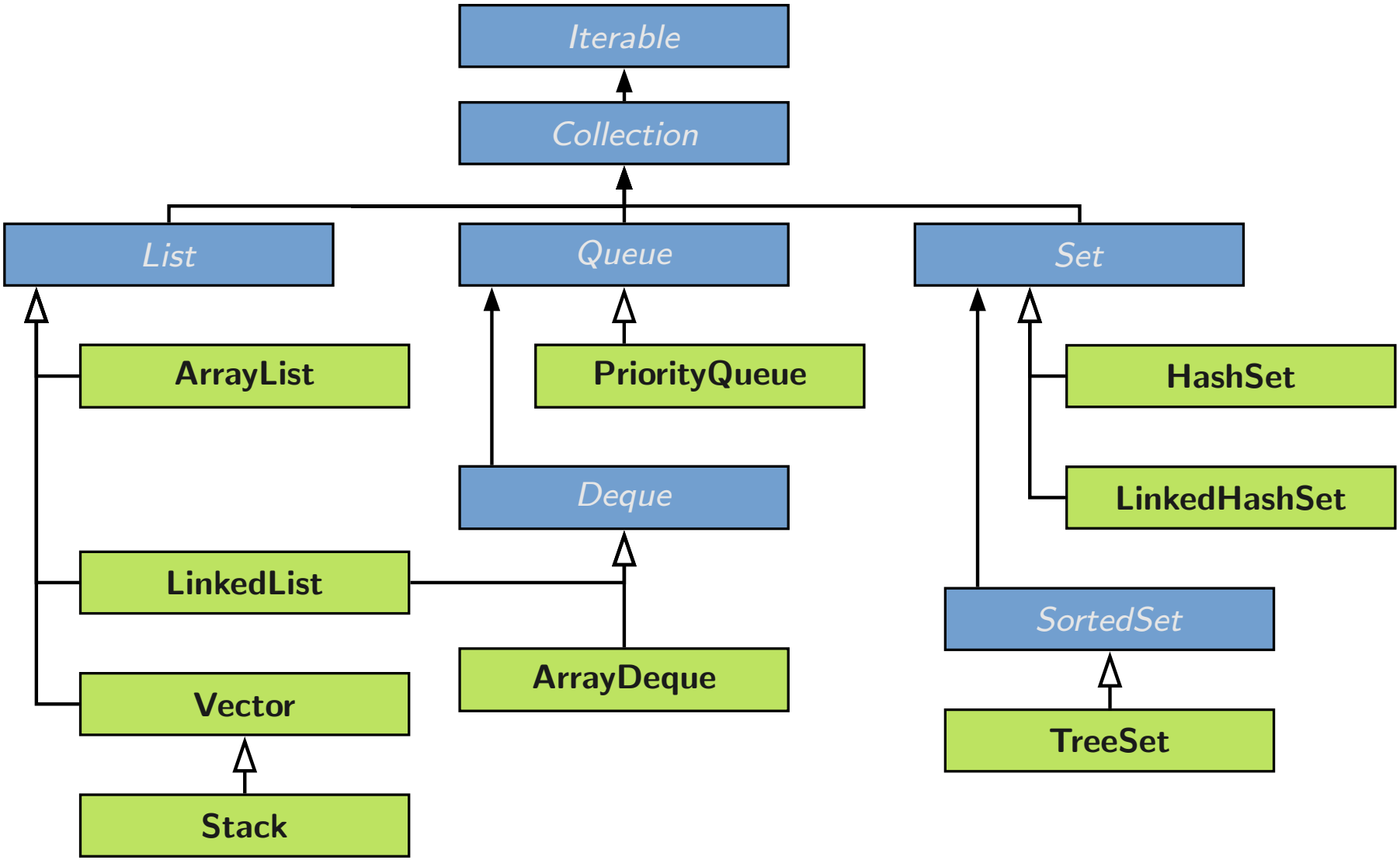
Bernd Kiefer
Jörg Steffen

December 2, 2022

Collections

- framework that provides an architecture to store and manipulate groups objects
- operations to manipulate data such as searching, sorting, insertion, manipulation, and deletion
- provides many interfaces (`Set`, `List`, `Queue`, `Deque`) and classes (`ArrayList`, `Vector`, `LinkedList`, `PriorityQueue`, `HashSet`, `LinkedHashSet`, `TreeSet`).

Collection Hierarchy



Iterator - Our first pattern

Why?

- access the elements of an aggregate object sequentially without exposing its underlying representation
- standard library abstraction that makes it possible to decouple collection classes and algorithms
- Promote to "full object status" the traversal of a collection
- implement the Iterator pattern
- "generic programming"

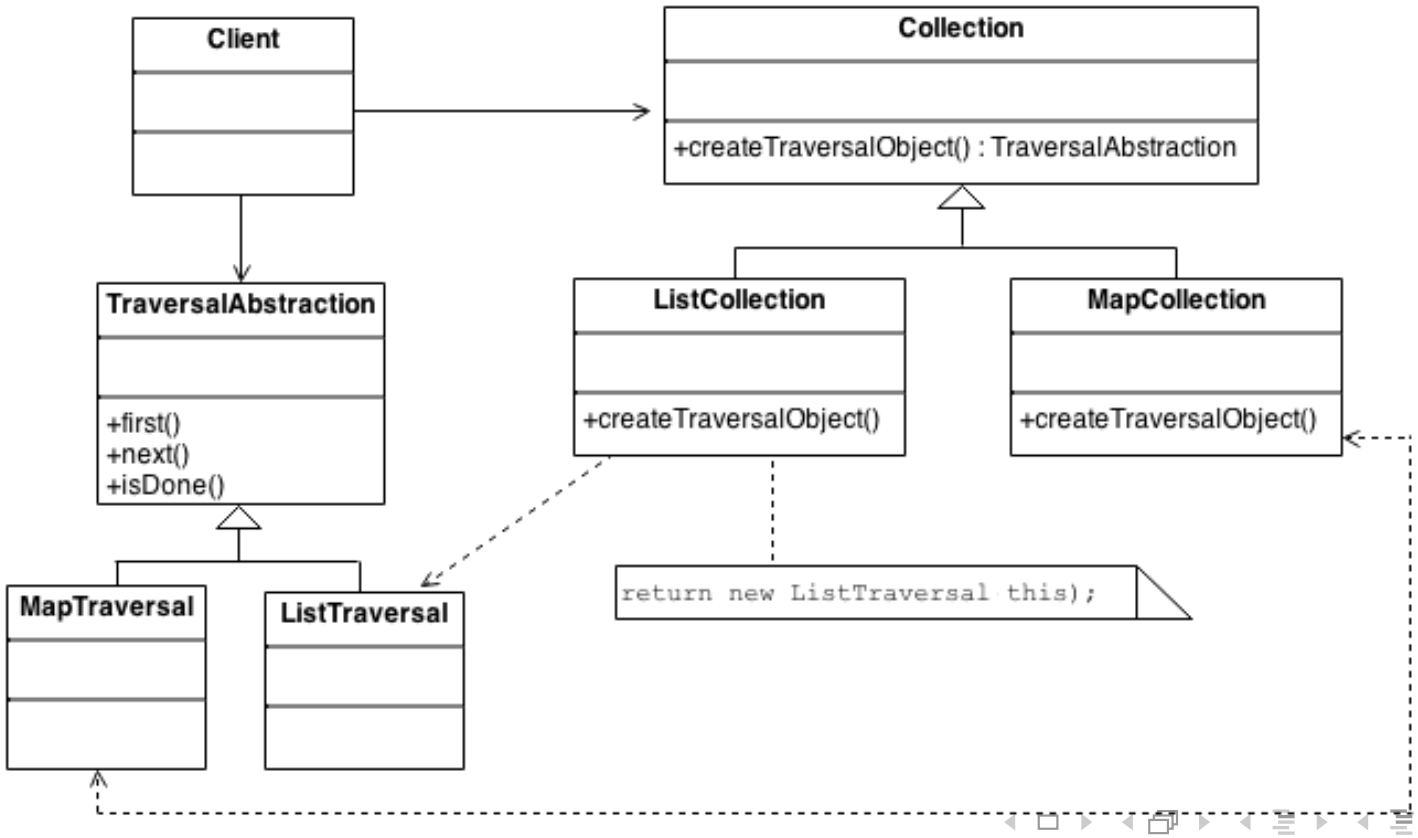
generic programming

- explicitly separate the notion of "algorithm" from that of "data structure")
- promote component-based development, boost productivity, and reduce configuration management

.

Iterator - Structure

- Access to the Collection's elements encapsulated behind additional level of abstraction called Iterator
- Collection derived class know which Iterator derived class to create
- Client relies on the interface defined in the Iterator base class



Iterator - How to?

Basic abstractions in the JRE

```
public interface Iterator {
  public boolean hasNext();
  public Object next();
}

public interface Iterable {
  public Iterator iterator();
}
```

Iterator - How to?

Create concrete class implementing interfaces.

```
public class NameRepository implements Iterable {
  @Override
  public Iterator iterator() {
    return new NameIterator();
  }

  private class NameIterator implements Iterator {
    @Override
    public boolean hasNext() { ... }

    @Override
    public Object next() { ... }
  }

}
```

Iterator - How to?

Step 3: Use the NameRepository to get iterator to print names

```
NameRepository namesRepository = new NameRepository();

for(Iterator iter = namesRepository.getIterator(); iter.hasNext();){
  String name = (String)iter.next();
  System.out.println("Name : " + name);
}

// or
Iterator iter = namesRepository.getIterator();
while (iter.hasNext()) {
  String name = (String)iter.next();
  System.out.println("Name : " + name);
}

// or simply
for(String name : namesRepository){
  System.out.println("Name : " + name);
}
```

Improving `Iterator` and `Iterable`

- Our `Iterator` and `Iterable` in the example are not so nice
- Any idea why? How could that be improved?

Improving Iterator and Iterable

- Our Iterator and Iterable in the example are not so nice
- Any idea why? How could that be improved?
- The Java way to do this: parameterized types (Generics)

```
public interface Iterator<T> {
  public boolean hasNext();
  public T next();
}

public interface Iterable<T> {
  public Iterator<T> iterator();
}
```

- We introduce a type parameter for our iterator

```
public class NameRepository implements Iterable<String> {
  @Override
  public Iterator<String> iterator() {
    return new NameIterator();
  }
  ...
```

Parameterised Types / Generics

- The JavaDoc of the Collections Framework contains definitions like: `interface List<E>`, `boolean add(E e)`, `E get(int index)`
- Allows to specify the type of elements in a collection:
```
List<String> l = new ArrayList<String>();
l.add("example");
```
- `List<E>` is a parameterised class type, with type parameter `E`
- You can define your own parameterised class: `public MyClass<E, F, G> { ... }`
- `E` can be used to specify variables, method parameters & return types
- However: no object construction of parameter type: ~~`new E()`~~

Defining Generic Classes (and Methods)

- In your class `MyClass<T>`, all the compiler knows is that `T` must be `Object` (no PODs allowed)
- So, you can at best call methods of `Object` on a `T` object
- But: you can also specify type constraints on the parameter: `MyClass<T extends Tile>` $\Rightarrow$ `T` is at least `Tile` (or subclass)
- Now, you can call `Tile` methods on an `T` object in `MyClass`
- There are also generic methods (also in non-generic classes):
  `<E extends Comparable<E>> List<E> sort(Collection<E> c)`

Wrapper Classes of PODs (int, double, boolean)

- We just learned that we can not store, e.g, an `int` in a `Collection`
- That's quite inconvenient

Wrapper Classes of PODs (int, double, boolean)

- We just learned that we can not store, e.g, an `int` in a `Collection`
- That's quite inconvenient
- The solution: wrapper classes, containing the value

`Integer`	$\leftrightarrow$	`int`
`Double`	$\leftrightarrow$	`double`
`Boolean`	$\leftrightarrow$	`bool`
`Character`	$\leftrightarrow$	`char`
etc.		

- All instances of these classes are immutable!
- Conversion (here: called (un)boxing) is in most cases automatic:

```
Integer integ = 0;
int i = integ + 1;
integ = integ + 1; // mark: this creates a new object!
```

- Used in, e.g, Maps: `Map<String, Integer> occurs = new HashMap<>();`

Modifying Functionality using Adapters

- In one of the excersises, you copied a list as result of a method to avoid manipulation of the internal state
- Actually, Java collections framework offers another way

The Immutable Pattern

An immutable version of a data structure is one where you can only read its contents, but not modify its state

```
public class Professor {
  private List<Student> students = new ArrayList<>();
  List<Student> getStudents() {
    return new ArrayList<Student>() {{ addAll(students); }};
  }
}
```

Modifying Functionality using Adapters

- In one of the excersises, you copied a list as result of a method to avoid manipulation of the internal state
- Actually, Java collections framework offers another way

The Immutable Pattern

An immutable version of a data structure is one where you can only read its contents, but not modify its state

```
public class Professor {
  private List<Student> students = new ArrayList<>();
  List<Student> getStudents() {
    return Collections.unmodifiableList(students);
  }
}
```

- An attempt to call, e.g., `add(elt)` on the result will result in an `UnsupportedOperationException`

The Adapter Pattern

- Adapter: Change the interface of a class
- Another `Collections` adapter: type safety

```
public class Professor {
  List<Student> checkAllStudents(List input) {
    return Collections.checkedList(input, Student.class);
  }
}
```

- If there is a non-student object in the list, this will throw a ClassCastException