Collections and Lambda

Object Oriented Programming
2024 First Semester
Shin-chi Tadaki (Saga University)

1. Collections
2. Utilities for collections and arrays
3. Maps
4. Streams
5. Lambda expressions

Today's sample programs

- https://github.com/oop-mc-saga/Lambda

Collections of instances

- Ordered objects:
  - List etc.
  - Queue: FirstIn-FirstOut
  - Stack: FirstIn-LastOut
- Set: not allow the same object to contain more than once
- Map: key-value pairs

Generic

- Definitions of classes and methods can contain parameterized target types. (not specify any concrete classes)
  - Collection classes have parameterized types indicating class instances contained in.
- When using a class with parameterized types
  - Compiler can detect type inconsistency, if parameterized types specified

Example 1.1: Student class

```
1   Student[] students = {
2       new Student("Tom", 1, 88),
3       new Student("Jane", 2, 80),
4       new Student("Ray", 3, 70),
5       new Student("Kim", 4, 75),
6       new Student("Jeff", 5, 85),
7       new Student("Ann", 6, 75),
8       new Student("Beth", 7, 90)
9   };
10  List<Student> studentList = new ArrayList<>();
11  for (Student s : students) {
12      studentList.add(s);
13  }
```

class parameter

omit

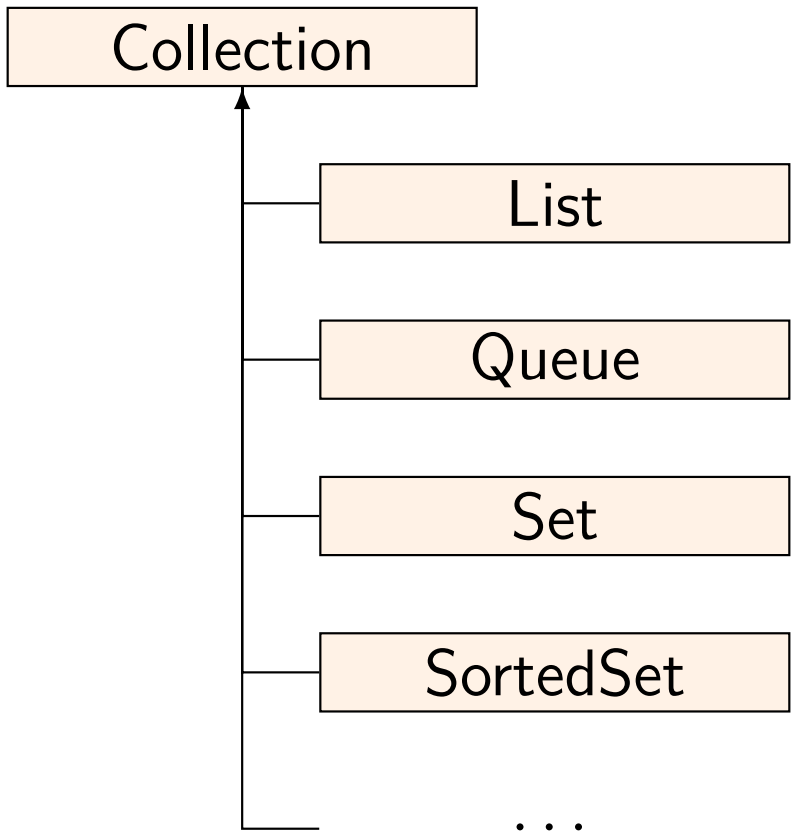
`studentList` is specified as a list of `Student` instances.

java.util.Collection

general

- The `Collection` is an interface for classes containing objects
- It has a type parameter for specifying class instances contained
- Major methods are as follows:
  - `boolean add()` : adds an element
  - `boolean contains()` : checks containing the specified element
  - `boolean isEmpty()` : checks the collection empty
  - `boolean remove()` : removes the specified element
  - `int size()` : returns number of elements
  - `Stream stream()` : returns stream for iterating elements

Collection and its extensions



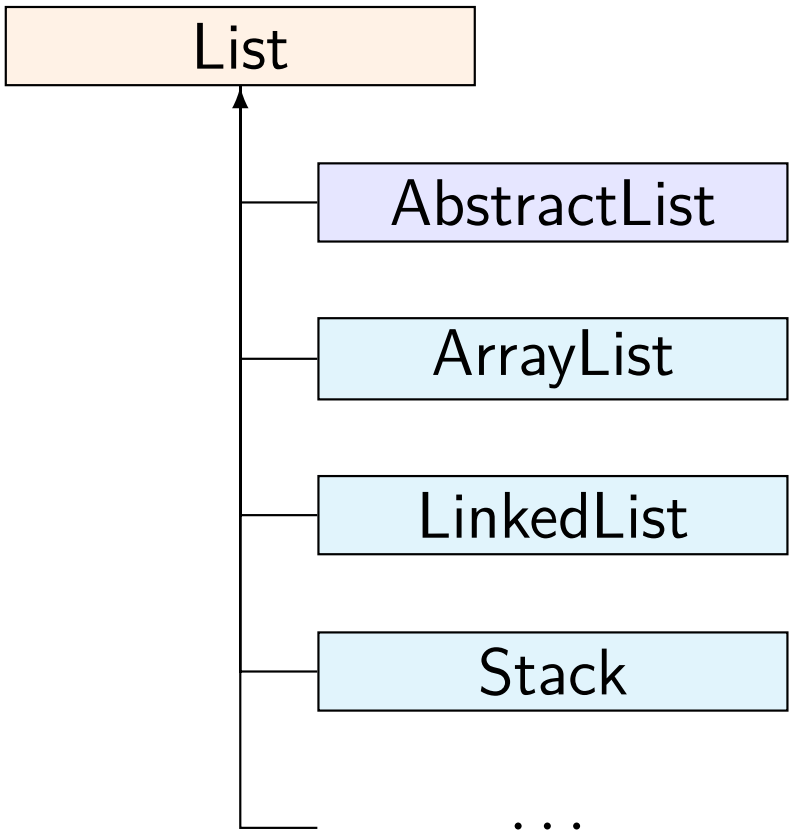
All are defined as ~~an~~ interface.

java.util.List

- List class stores ordered elements
- Major methods
  - `boolean add()`: adds an element at the end. Throw exception if unsuccess.
  - `E get()` : returns the element at the specified position
  - `int indexOf()` : returns the position of the specified element
  - `E set()` : sets the element at the specified position and returns the element.

remove() : Remove the specified element

Implementations of `java.util.List`



Quick access to elements

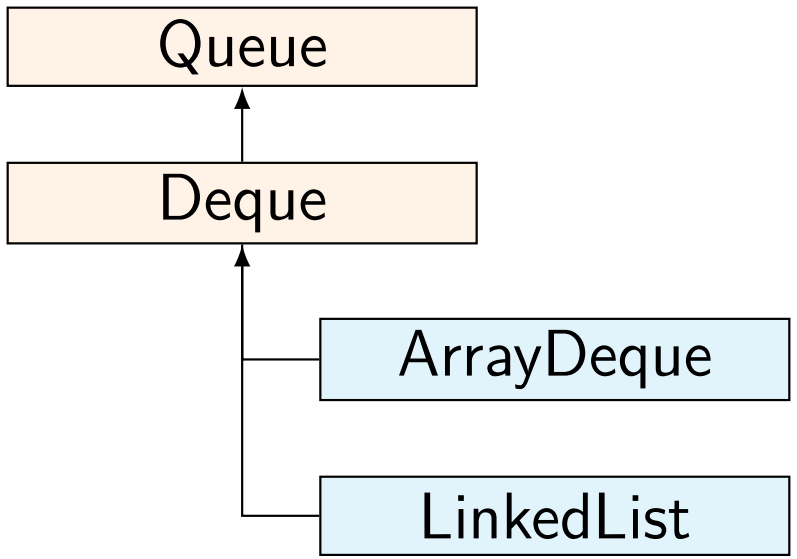
Quick add and remove elements

Example of list operation

java.util.Deque

- Double ended queue
- *Queue* allows to operate elements at the ends of the list
  - Queue: FirstIn-FirstOut
  - Stack: FirstIn-LastOut
- Major methods
  - `offerLast(e)`: adds an element at the tail
  - `pollLast()`: removes the element at the tail and return it
  - `pollFirst()`: removes the element at the head and return it

Implementations of `java.util.Deque`

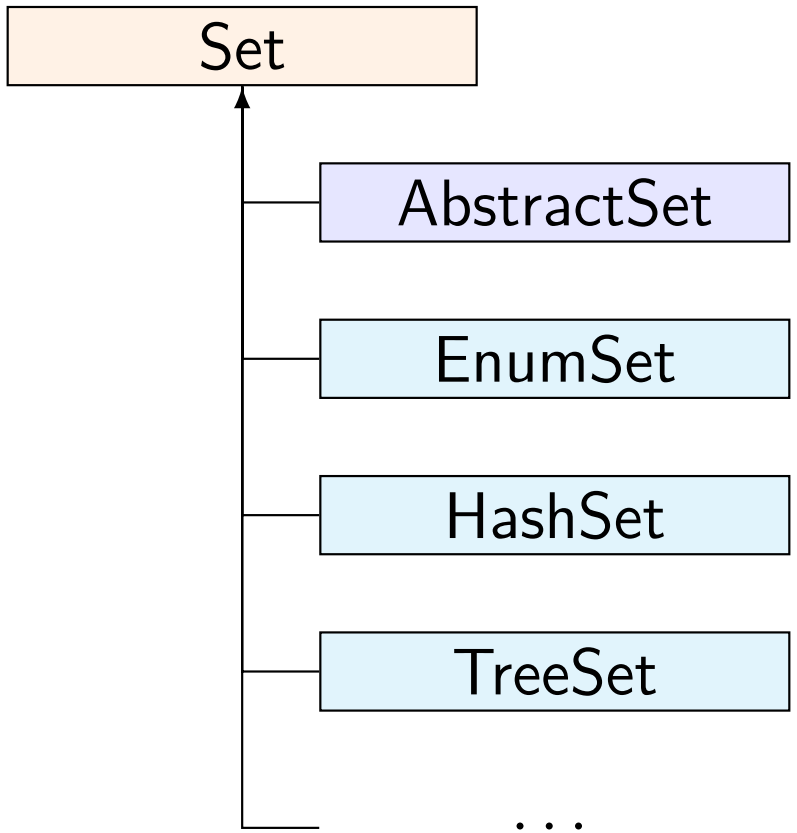


Example of Deque operations

java.util.Set

- Set stores elements and not allows the same element to contain more than once.
  - Similarity is decided by `equals()` method of the element class
- Major methods
  - `contains()`: returns whether the set contains the specified element or not.
- The order of elements are indeterminate.

Implementations of `java.util.Set`



Examples of see operation

`Collections` class
Methods for operate collections

- search element
- maximum and minimum element
- reverse order
- thread protection
- sort
- swap elements
- protecting modification

See `Lambda.collectionsSample`

```java
//Search element in list
int k = Collections.binarySearch(studentList, students[3]);
System.out.println(students[3] + " is found at " + k);

//Find the maximum element
Student best = Collections.max(studentList);
System.out.println(best + " marks the best");

//Sort list
System.out.println("sorted list");
Collections.sort(studentList);
studentList.forEach(
        s -> System.out.println(s)
);
System.out.println("----------------");
//Copy list to array
Student[] studentArray = new Student[studentList.size()];
studentArray = studentList.toArray(studentArray);
for (Student s : studentArray) {
    System.out.println(s);
}
System.out.println("----------------");

//Create immutable view of list
List<Student> view = Collections.unmodifiableList(studentList);
try {
    Collections.reverse(view);
} catch (UnsupportedOperationException e) {
    System.err.println("This list is immutable.");
}
```

`Arrays` class
methods for operating arrays

- convert to list
- search element
- copy array
- compare arrays
- sort
- convert to string

See `Lambda.arraysSample`

java.util.Map

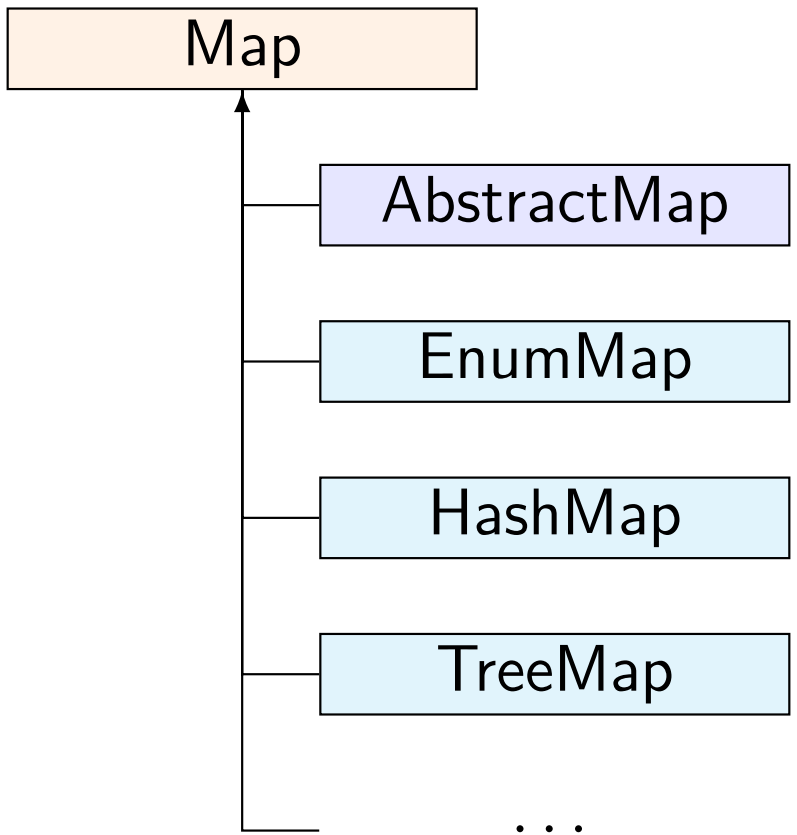
- Map class stores Key-Value pairs
- Major methods
  - `V get()`: returns value specified by a key
  - `Set<K> keySet()`: returns a set of key
  - `V put()`: put a key-value pair. The value is update if the key exists.
  - `Collection<V> values()`: returns a collection of values.

See `Lambda.mapSample`

Example

```
1  public static void main(String[] args) {
2      String codes[] = {"CTS", "FUK", "HSG", "HND", "KIX"};
3      String names[] = {"Sapporo (New Chitose)", "Fukuoka", "Ariake
       ↪  Saga",
4          "Haneda", "Kansai"};
5
6      Map<String, String> airports = new HashMap<>();
7      for (int i = 0; i < codes.length; i++) {
8          airports.put(codes[i], names[i]);
9      }
10
11     for (String code : airports.keySet()) {
12         System.out.println(code + "->" + airports.get(code));
13     }
14 }
```

Implementations of `java.util.Map`



Threads and collections

- We need to prevent multiple threads from simultaneous accesses to collections.
  - Simultaneous attempts for modifying a collection ~~will~~ may induce inconsistency and destroy the target.
- For protecting, use
  - `Collections.synchronizedList()`
  - `Collections.synchronizedSet()`
  - etc.
  - Or use *blocking* classes

example

Operation for all elements in a collection

- Extended *for* loops

```
1  List<T> list;
2  for ( T t: list){
3      do something on t
4  }
```

- Using `Stream` and Lambda expressions

java.util.stream.Stream

- A sequence of element
  - sequential and parallel operations
- Major methods
  - `Stream<T> filter()`: filters elements by a predicate
  - `void forEach()`: performs an operation for each element
  - `void forEachOrdered()`: performs an operation for each element in the order of the stream
  - `Optional<T> reduce()`: Performs a reduction on the elements
  - Arguments are instances of classes in `java.util.function` package.

See `Lambda.lambdaSamples`

```java
public static void main(String[] args) {
    int n=100;
    List<Double> list = new ArrayList<>();
    for(int i=0;i<n;i++){
        list.add(Math.random());
    }
    //print all elements
    list.stream().forEach(d ->  System.out.println(d));
}
```

- The argument of `forEach()` is an instance of `Consumer` interface.
  - It accepts one argument and performs an operation without returning any value.

Without Lambda

```
1  public static void main(String[] args) {
2      int n = 100;
3      List<Double> list = new ArrayList<>();
4      for (int i = 0; i < n; i++) {
5          list.add(Math.random());
6      }
7      Consumer<Double> c = new Consumer<>(){
8          @Override
9          public void accept(Double d){
10             System.out.println(d);
11         }
12     };
13     //print all elements
14     list.stream().forEach(c);
15 }
```

Lambda expressions

- A lambda expression defines an anonymous method.
- It enables us to treat a function as an argument of methods.
- Lambda expressions use interface mechanisms in Java
- Various typical functions are defined in `java.util.function`
  - The `apply()` method is defined in those interfaces.

Fundamentals of Lambda expressions

- Fundamental notation

```
(arguments) -> {operation}
```

  - type of arguments can be omitted
  - `()` can be omitted for one argument case
  - `{}` can be omitted for one-line operation

Examples of `java.util.function`

- `BinaryOperator<T>`
  - operation upon two operands of the same type, producing a result of the same type
- `DoubleBinaryOperator`
  - operation upon two `double` operands, producing a result of `Double`
- `DoubleFunction<R>`
  - operation upon one `double` operand, producing a result of `R`

Example 5.1: `listOperation()`

```
1  public static List<Integer> listOperation(List<Integer> inputList,
2          IntFunction<Integer> func) {
3      List<Integer> outputList = new ArrayList<>();
4      for (int i = 0; i < inputList.size(); i++) {
5          int input = inputList.get(i);
6          int output = func.apply(input);
7          outputList.add(output);
8      }
9      return outputList;
10 }
```

main method

```
1  public static void main(String[] args) {
2      List<Integer> inputList = new ArrayList<>();
3      for (int i = 0; i < 5; i++) {
4          inputList.add(i);
5      }
6      List<Integer> outputList = listOperation(inputList,
7          x -> x * x
8      );
9      outputList.forEach(
10         x -> System.out.println(x)
11     );
12 }
```

Exercise

Pass a lambda expression for squared sum in `sumAll()` method.