Object-Oriented Analysis & Java

Lecture 5

Florentin Bekier

Generics

Need for generics

- The idea is to allow type (`Integer`, `String`, etc and user defined types) to be a parameter to methods, classes and interfaces
- Classes like `HashSet`, `ArrayList`, `HashMap`, etc use generics very well

Generic class

- We use the diamond operator <> to specify parameter types in generic class creation

```
class Test<T> {
    T obj;
    Test(T obj) {
        this.obj = obj;
    }
    public T getObject() {
        return this.obj;
    }
}
```

Generic class

- To create an instance of a generic class we use the following syntax:

```
Test<Integer> obj = new Test<Integer>(5);
```

Generic methods

- We can also write generic functions that can be called with different types of arguments based on the type of arguments passed to generic method

```
static <T> void genericDisplay(T element) {
    System.out.println(element.getClass().getName() + "
= " + element);
}
```

Advantages

- Code reuse: We can write a method/class/interface once and use for any type we want
- Type safety: Generics make errors to appear compile time than at run time
- Individual type casting is not needed

Collections

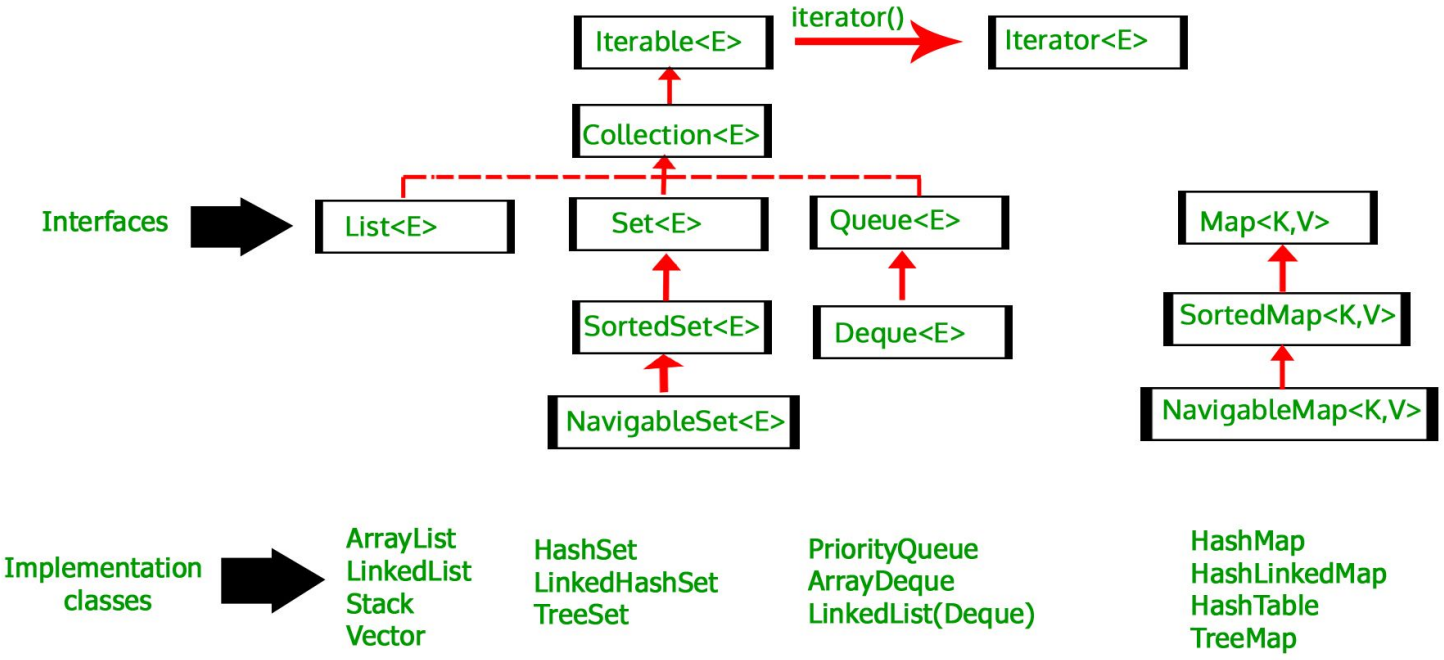
Collections in Java

- Groups of individual objects represented as a single unit
- Collection Framework: defines several classes and interfaces to represent collections in Java
- The Collection interface (`java.util.Collection`) and Map interface (`java.util.Map`) are the two main “root” interfaces of Java collection classes

Common interface

- `add(Object o)`: Add an object to the collection
- `remove(Object o)`: Remove an object from the collection, if it exists
- `clear()`: Remove all the objects from the collection
- `contains(Object o)`: Return true if the collection contains the object
- `size()`: Return the number of elements in the collection
- `isEmpty()`: Return true if the collection is empty, false otherwise
- `iterator()`: Return an iterator to browse the content of the collection

Hierarchy


Iterable<E>
iterator()
Iterator<E>
Collection<E>
Interfaces
List<E>
Set<E>
Queue<E>
Map<K,V>
SortedSet<E>
Deque<E>
SortedMap<K,V>
NavigableSet<E>
NavigableMap<K,V>
Implementation classes
ArrayList
LinkedList
Stack
Vector
HashSet
LinkedHashSet
TreeSet
PriorityQueue
ArrayDeque
LinkedList(Deque)
HashMap
HashLinkedMap
HashTable
TreeMap

List

- Can contain duplicates and elements are ordered
- Example implementations: `LinkedList` (linked list based) and `ArrayList` (dynamic array based)
- Specific methods: `add(int index, Object o)`, `get(int index)`, `remove(int index)`, `indexOf(Object o)`

Set

- Doesn't allow duplicates
- Example implementations: `HashSet` (Hashing based) and `TreeSet` (balanced BST based)
- `TreeSet` implements `SortedSet`

Map

- Contains key-value pairs
- Doesn't allow duplicates
- Example implementation: `HashMap` and `TreeMap`
- `TreeMap` implements `SortedMap`

Map methods

- `get(Object key)`: Get the value associated to the key
- `put(Object key, Object value)`: Add a key-value pair
- `remove(Object key)`: Remove the pair associated to the key
- `containsKey(Object key)`: Check the existence of a key
- `containsValue(Object value)`: Check the existence of a value
- `values()`: Return a Collection of values
- `keySet()`: Return a Set of keys

File handling

Files

- The `File` class from the `java.io` package, allows us to work with files
- To use the `File` class, create an object of the class, and specify the filename or directory name:

```
import java.io.File;  // Import the File class

File myObj = new File("filename.txt"); // Specify the filename
```

Methods

- `canRead()`: Tests whether the file is readable or not
- `canWrite()`: Tests whether the file is writable or not
- `createNewFile()`: Creates an empty file
- `delete()`: Deletes a file
- `exists()`: Tests whether the file exists

Methods

- `getName()`: Returns the name of the file
- `getAbsolutePath()`: Returns the absolute pathname of the file
- `length()`: Returns the size of the file in bytes
- `list()`: Returns an array of the files in the directory
- `mkdir()`: Creates a directory

Read a file

```
public class ReadFile {
     public static void main(String[] args) {
          try {
               File myObj = new File("filename.txt");
               Scanner myReader = new Scanner(myObj);
               while (myReader.hasNextLine()) {
                    String data = myReader.nextLine();
                    System.out.println(data);
               }
               myReader.close();
          } catch (FileNotFoundException e) {
               System.out.println("An error occurred.");
               e.printStackTrace();
          }
     }
}
```

Write to a file

```
public class WriteToFile {
     public static void main(String[] args) {
          try {
               FileWriter myWriter = new FileWriter("filename.txt");
               myWriter.write("Files in Java might be tricky, but it is fun
          enough!");
               myWriter.close();
               System.out.println("Successfully wrote to the file.");
          } catch (IOException e) {
               System.out.println("An error occurred.");
               e.printStackTrace();
          }
     }
}
```