Computer Science 22: Object Oriented Programming

Lecture #17: On Persistence and Serialization; Some Useful Classes

In This Lecture

- Persistence
- Serialization
- Java Collections Interface
  - Collection
  - Set
  - List
  - Map

Persistence

- Objects may outlive the context they are created in. They can be “saved” and then “reloaded” when we again have a new session for its use.
- Persistence can be achieved by storing the state of the object in non-volatile storage such as hard drive.
- Objects should be serialized first before it can be saved into a file.

Serialization

- Process of converting an **object state** into a format that can be stored and reloaded later.
- Serialization of objects does not include their associated methods.
- The process of serializing a method is also called deflating or marshalling an object.
- The reverse operation, extracting a data structure from a series of bytes, is deserialization (also called inflating or unmarshalling an object).

Serialization

- A Java class should implement the interface java.io.Serializable for its instance states' to be convertible to bytes that can be saved to a persistent storage.

```
public class Student implements Serializable {
    /* attributes and methods of the class */
}
```

Serialization

- We can use specific stream classes to read from and write into a file the state of an object.
  - ObjectOutputStream (writing)
  - ObjectInputStream (reading)

Serialization

```
Student s = new Student(“2004-95317”, “JUAN DELA CRUZ”);
try {
    ObjectOutputStream out = new ObjectOutputStream(new
        FileOutputStream(“students.dat”, true));
    out.writeObject(s)
    out.close();
}catch(Exception e) {}
```

Serialization

```
try {
    ObjectInputStream in = new ObjectInputStream (new
        FileInputStream(“students.dat”));
    Student s = (Student) in.readObject()
    in.close();
}catch(Exception e) {}
```

Java Collections Interface

- Collection
- Set
- List
- Map

Collection

- A Collection represents a group of objects known as its elements.
  - int size();
  - boolean isEmpty();
  - boolean contains(Object element);
  - boolean add(E element);
  - boolean remove(Object element);
  - Iterator<E> iterator();

  - boolean containsAll(Collection<?> e);
  - boolean addAll(Collection<?> e);
  - boolean removeAll(Collection<?> e);
  - boolean retainAll(Collection<?> e);
  - void clear();

  - Object[] toArray();
  - <T>T[] toArray(T[] a);

Set

- A collection that contains no duplicate elements.
  - int size();
  - boolean isEmpty();
  - boolean contains(Object element);
  - boolean add(E element);
  - boolean remove(Object element);
  - Iterator<E> iterator();

  - boolean containsAll(Collection<?> e);
  - boolean addAll(Collection<?> e);
  - boolean removeAll(Collection<?> e);
  - boolean retainAll(Collection<?> e);
  - void clear();

  - Object[] toArray();
  - <T>T[] toArray(T[] a);

List

- An ordered collection that may contain duplicate elements.
  - E get(int index);
  - E set(int index, E element);
  - boolean add(E element);
  - void add(int index, E element);
  - E remove(int index);
  - boolean addAll(int index, Collection<? Extends E> c);

  - int indexOf(Object o);
  - int lastIndexOf(Object o);

  - ListIterator<E> listIterator();
  - ListIterator<E> listIterator(int index);

  - List<E> subList(int from, int to);

Map

- An object that associates keys to values. A map cannot contain duplicate keys and each key can only be assigned to one value.
  - V put(K key, V value);
  - V get(K key);
  - V remove(Object key);
  - boolean containsKey(Object key);
  - boolean containsValue(Object value);
  - int size();
  - boolean isEmpty();

  - void putAll(Map<? Extends K, ? Extends V> m);
  - void clear();

  - Set<K> keySet();
  - Collection<V> values();
  - Set<Map.Entry<K, V>> entrySet();