Java Lang Review, Part III

Nested Classes

Java Collections

Exceptions

Jianjun Chen (Jianjun.Chen@xjtlu.edu.cn)

Nested Classes

Static/Non-static nested Classes, local classes, anonymous classes

Nested Classes

- Java allows you to define a class within another class. Such a class is called a nested class.
- Nested classes are divided in to four categories:
  - **Static**: member classes declared as static.
  - **Non-static**: member classes not declared as static, also know as inner classes.
  - **Local**: classes declared in the body of a function.
  - **Anonymous**: local classes automatically declared and instantiated in the middle of an expression.

Static Nested Classes

```
class A {
    static class B {
    }
}
```

- Defined along with the outer class.
- Has similar access properties like other static members of the outer class.
  - A static nested class cannot refer directly to the instance variables or methods defined in its enclosing class.
  - But it can refer to static variables and methods.
- Accessed using the enclosing class name:

```
OuterClass.StaticNestedClass nestedObject =
        new OuterClass.StaticNestedClass();
```

Static Nested Classes

kitfox — Vim A.java — 80×26

```
public class A {
        static class B {
                public static void bs() {
                        A.as();
                        System.out.println("b static");
                }
                public void bi() {
                        System.out.println("b instance");
                }
        }

        public static void as() {
                System.out.println("a static");
        }
        public void ai() {
                System.out.println("a instance");
        }

        public static void main(String[] args) {
                A.B b = new A.B();
                A.B.bs();
                b.bi();
        }
}
~
                                                        11,0-1        All
```

Non-static Nested Classes

```
class A {
    class B {
    }
}
```

- Inner class is associated with an instance of its enclosing class and has direct access to that object's methods and fields.
  - You must create an object of the outer class first.

- Inner class cannot have static members.
  - It can have `static final` variables though.

Non-static Nested Classes

```
class A {
    class B {
    }
}
```

- Objects that are instances of an inner class exist within an instance of the outer class.

```
OuterClass outerObject = new OuterClass();
OuterClass.InnerClass innerObject =
            outerObject.new InnerClass();
```

Non-static Nested Classes

kitfox — Vim A.java — 80×26

```
public class A {
        class B {
                public void bi() {
                        A.as();
                        A.this.ai();
                        System.out.println("b instance");
                }
        }

        public static void as() {
                System.out.println("a static");
        }
        public void ai() {
                System.out.println("a instance");
        }

        public static void main(String[] args) {
                A a = new A();
                A.B b1 = a.new B();
                A.B b2 = a.new B();
                b1.bi();
        }
}
~
~
                                                        9,0-1         All
```

Nested Classes: Shadowing

- https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html

```
public class A {
    int x = 1;
    class B {
        int x = 2;
        void testX(int x) {
            System.out.printf("%d, %d, %d\n", x, this.x, A.this.x);
        }
    }
    public static void main(String[] args) {
        A a = new A();
        A.B b = a.new B();
        System.out.printf("%d, %d\n", a.x, b.x);
        b.testX(3);
    }
}
```

Question

- When are the static final members of inner classes initialised?

- Can you write a few lines of code to prove your point?

Variants of Inner Class

- There are three variants of inner class:
  - Local classes
  - Anonymous classes
  - Lambda expressions

- Anonymous classes are local classes without class names.
- Lambda expressions are anonymous classes with only one function.

Local Classes

- Local classes can be defined in any blocks { … }.
- Local classes can only access `final` local variables of the function.

```
public void func3() {
    class LocalClass{
        public void localMethod(){
            System.out.println("this is from the local class");
        }
    }
    LocalClass localClass = new LocalClass();
    localClass.localMethod();
}
```

Local Classes

- Local classes are similar to inner classes:
  - Cannot have static members.
- Local classes in static methods can only refer to static members of enclosing classes.
- If in non-static methods, they can access non-static members of the enclosing classes.

Anonymous Classes

- An anonymous class is a local class without a name.
- It is defined and instantiated (at the same time) in a single expression using the `new` operator.

```
public class AnonymousTest {
    int x = 0;
    static int y = 0;
    public void f() {
        Button btn = new Button();
        btn.setText("Say 'Hello World'");
        btn.setOnAction(new EventHandler<ActionEvent>() {

            @Override
            public void handle(ActionEvent event) {
                x = 0;
                y = 0;
                System.out.println("Hello World!");
            }
        });
    }
```

Anonymous Classes

- An anonymous class must either
  - Extend another class
  - Or implement an interface
- Format:

```
new constructor_name(parameters) {
      //inner class methods and variables
}
```

The constructor name is the same as the superclass name or the interface name.

*** You **cannot** define your own constructors.

Another example

```
public class AnonymousClass {
    static class MySuperclass {
        int a;
        MySuperclass(int a) {
            this.a = a;
        }
        void print() {
            System.out.println(a);
        }

    }

    public static void main(String[] args) {
          takeObject(new MySuperclass(5) {
              void print() {
                  System.out.println("subclass: " + a);
              }
          });
    }

    public static void takeObject(MySuperclass msc) {
        msc.print();
    }
}
```

Why Using Nested Classes?

- It is a way of logically grouping classes that are only used in one place.
  - leads to more readable and maintainable code

- It increases encapsulation

"Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world."

https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html

Collections

List: ArrayList, LinkedList

Set: HashSet, TreeSet, LinkedHashSet

Map: HashMap

ArrayList

- Internal storage is array.
- Extends its capacity once full.
  - Create a new array then copy the old array's content.

- Elements of ArrayList must be objects.
  - Primitive types like `int`, `double`, `boolean` are not allowed
  - But you can use wrapper classes `Integer`, `Double`, `Boolean`.
  - E.g. `arrayList.add(new Integer(5));`

LinkedList

- The LinkedList class provides an implementation of a list that is based on linked nodes.
- Because the node links must be traversed, access to the list's elements is slow.
- Because only node references need to be changed, insertions and deletions of elements to the LinkedList are fast.

LinkedList: Question

- Is the following implementation efficient?
- If no, which part is slow?

```
LinkedList<Integer> intList = new LinkedList<Integer>();
// add elements to list
for (int i = 0; i < intList.size(); i++) {
    System.out.println(intList.get(i));
}
```

```java
LinkedList<Integer> intList = new LinkedList<Integer>();
// add elements to list
for (int i = 0; i < intList.size(); i++) {
    System.out.println(intList.get(i));
}
```

```java
/**
 * Returns the number of elements in this list.
 *
 * @return the number of elements in this list
 */
public int size() {
    return size;
}
```

Not all list implementations iterate through the whole list to get the size. Check the documentation.

```java
LinkedList<Integer> intList = new LinkedList<Integer>();
// add elements to list
for (int i = 0; i < intList.size(); i++) {
    System.out.println(intList.get(i));
}
```

```java
public E get(int index) {
    checkElementIndex(index);
    return node(index).item;
}
```

```java
/**
 * Returns the (non-null) Node at the specified element index.
 */
Node<E> node(int index) {
    // assert isElementIndex(index);

    if (index < (size >> 1)) {
        Node<E> x = first;
        for (int i = 0; i < index; i++)
            x = x.next;
        return x;
    } else {
        Node<E> x = last;
        for (int i = size - 1; i > index; i--)
            x = x.prev;
        return x;
    }
}
```

Use iterators instead

Set

- Set stores items like List and Array, except that it does not allow duplicate elements.

- There are three general-purpose implementations: `HashSet`, `TreeSet`, and `LinkedHashSet`.
  - `HashSet` is much faster than `TreeSet`, provides constant-time for most operations but offers no ordering.
  - If you need ordering operations then use `TreeSet`.
  - `LinkedHashSet` is intermediate between `HashSet` and `TreeSet`.

Set: Duplication Detection

- The detection is achieved using `equals()` and `hashCode()` method from Object class.
  - If you want to add instances of your own classes to a Set. Your classes must override both functions.
  - Otherwise, duplicate class instances can be stored in the Set.
- For primitive data types (e.g. `int`) you do not need to override the equals method.
  - They are implicitly converted into wrapper classes like `Integer` first.
  - These wrapper classes provide these overrides.

Java API: Integer Class

hashCode

```
public int hashCode()
```

Returns a hash code for this `Integer`.

Overrides:

`hashCode` in class `Object`

Returns:

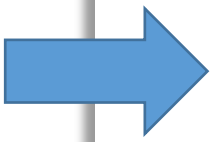
a hash code value for this object, equal to the primitive int value represented by this Integer object.

See Also:

`Object.equals(java.lang.Object)`, `System.identityHashCode(java.lang.Object)`

(Hash)Set: Example 2

The second A(2) will be added to the HashSet because it is not a primitive data type, Java would consider them two different objects



```
class A {
    public int x;
    A(int x) { this.x = x; }
}

public class SetTest {
    public static void main(String[] args) {
        HashSet<A> aset = new HashSet<A>();
        aset.add(new A(2));
        aset.add(new A(2));
        for(A a : aset) {
            System.out.println(a.x);
        }
    }
}
```

(Hash)Set: Example 2

No more duplications in this example:

Output has only one "2"

```
import java.util.HashSet;

class A {
    public int x;
    A(int x) { this.x = x; }

    public int hashCode() {
        return x;
    }

    public boolean equals(Object obj) {
        return obj != null
                && obj.getClass().equals(getClass())
                && this.x == ((A)obj).x;
    }
}

public class SetTest {
    public static void main(String[] args) {
        HashSet<A> aset = new HashSet<A>();
        aset.add(new A(2));
        aset.add(new A(2));
        for(A a : aset) {
            System.out.println(a.x);
        }
    }
}
```

Map

- `HashSet` internally use `HashMap`.
- A map is a group of key/value pairs (aka entries).
  - generic type is `Map<K, V>` (`K` is the key's type; `V` is the value's type).
- The key identifies an entry, a map cannot contain duplicate keys.
- Furthermore, each key can map to at most one value, e.g,
  - the key can be a String specifying student ID
  - the value can be the student's object which includes his/her email

HashMap: Example

Valid, but don't write in this way.

```
//HashMap ids = new HashMap();
HashMap<Integer, String> ids =
        new HashMap<Integer, String>();
ids.put(1001, "Name1");
ids.put(1002, "Name2");
ids.put(1003, "Name3");
```

Exceptions

- Using superclass "`Exception`" to catch all exceptions is not a good practice.
- You should deal with each case of exception, separately.
  - So that your code can achieve higher robustness.

```
try{
   //code where exception might occur;

} catch(exception object){
   //code to handle exception;

} catch(another exception object){
   //code to handle exception;

} finally{
   //cleaning code always executed

}
```

Exception Types

- Error: Indicates serious problems that a reasonable application should not try to catch.
- Exception: They are checked exceptions
  - Must be handled by `try/catch` block explicitly.
  - Or add a throws clause the the method.
  - E.g. `java.io.FileNotFoundException`
- RuntimeException: RuntimeException and its subclasses are unchecked exceptions.
  - Do not need to be handled by the calling code.
  - E.g. `NullPointerException`