Tutorial 7

Exceptions

Exceptions

- 3 main ways of handling errors - returning error codes, assertions, and exceptions
- Exceptions are abnormal events that happen during the course of a program
- Example - opening a nonexistent file

- In Java, certain (checked) exceptions must be handled by your code in some way

Exception Handling

- Three main operations - declaring exceptions, throwing exceptions, and catching exceptions
- Latter 2 are methods of *handling exceptions*

Declaring Exceptions

- Methods must declare the exceptions they could possibly throw
- Use the **throws** keyword in the method signature

```
public void methodA() throws ExceptionX {
   // An abnormal condition inside this method body may trigger ExceptionX }
```

Handling Exceptions

- When you call a method which declares an exception, you must handle the error in some way
- You must either:
  - Catch the exception and handle it using **try - catch**
  - Redeclare the exception in your own method and throw it up the calling chain (“not my problem”)

Catching Exceptions

```
public class ScannerFromFile {
        public static void main(String[] args) {
           Scanner in = null;
           try {
             in = new Scanner(new File("input.txt"));
           } catch (FileNotFoundException e) {
             e.printStackTrace();
           } finally {
             in.close();
           }
        }
}
```

Re-Throwing Exception

```
public class ScannerFromFile {
        public static void main(String[] args) throws FileNotFoundException {
           Scanner in = new Scanner(new File("input.txt"));
           // do something ...
        }
}
```

LSP & Exceptions

- Suppose in parent class, method() throws ExceptionA
- In the child class, can method() throws ExceptionA, ExceptionB, where ExceptionB is not a subclass of ExceptionA?

Miscellaneous OOP Concepts that are relevant to the Lab

Variance

- Variance is a concept in OOP which relates to how subtyping of simple components affects subtyping of complex / composite types
- Covariance - composite type preserves ordering of component types
- Contravariance - composite type reverses ordering of component types
- Invariance - no ordering of types (all types unequal to each other)

Variance - Arrays

- Suppose we have an **Animal** class, and a **Cat** class: **Cat < Animal**

- If Arrays (the composite) were covariant: an array of Cats is an array of Animals (**Cat[] < Animal[]**)
- If Arrays were contravariant: an array of Animals is an array of Cats (**Animal[] < Cat[]**)
- If Arrays were invariant: an array of Cats not an array of Animals, an array of Animals is not an array of Cats
- In Java, arrays are **covariant**

Collections

- A collection refers to a general group of objects
- For example, a Set, or a ArrayList, or a Bag/Multiset

- We covered before the practice of using an interface to define how a Collection Behaves (ShapeBag)
- Question - how can we extend Bag to work with objects of any type (including custom classes)?

Parameterized Types (Generics)

- Solution - accept the type associated with an object as a parameter
- Define an interface with works with a *generic* type **E**
- Clients can they specify what type they want to work with (eg, Integer)

- Seen in ArrayList - when you specify ArrayList<Hotel>, for example

Iterator Pattern

- Motivation - you have a class which aggregates many objects and you expect clients to want to iterate through all the items in your class
- You also want your class to work with generic searching algorithms

- You don’t want to expose the internal data structure to Clients, nor do you want to define a customer iteration interface

Iterator Interface

- Java defines a standard Iterator interface, with methods such as **next** and **hasNext**
- Allows you to define your own iterator classes
- Passing your custom Iterator class to methods which receive Iterator types will allow them to iterate over your class without having to know about the internal representation

Iterable Interface

- Problem with Iterator interface - we still needed a custom interface method, such as createIterator(), to retrieve the iterator
- Client still needs to know a little about your implementation

- We want them to be iterate over your collection knowing absolutely nothing (other than the fact it is iterable over)
- Enter the **Iterable** interface

Iterable Interface

- Interface with single (required) method - **iterator()**
- Returns an iterator

- Enables clients to use the foreach syntax (for (x : y) {})
- Clients don’t need to know how to retrieve your iterator