Generics and Using a Collection

- Generics / Parameterized Classes
- Using a Collection
- Customizing a Collection using Inheritance
- Inner Classes
- Use of Exceptions
- Reading: L&C: 3.2-3.3, 3.5
Java Generics

• In Java 5.0, Sun added a generic feature that had been available in other languages such as C++ (where it is called ‘Templates’)
• Although the implementation is different, the user gains the benefit of stronger compiler type checking and simpler coding style (with a reduced number of explicit casts needed)
Java Generics

- The Java implementation was constrained by backward compatibility
- Code compiled using javac 5.0 and generics had to work on java 4.2 virtual machines
- The technique used was “type erasure” and one copy of the parameterized class code is accessed with compiler supplied casts
- Avoids C++ “code bloat” where there are multiple copies of the parameterized code
Parameterized Classes

- Defining a parameterized class:

```java
public class Generic<T> { 
    // use T in attribute declarations
    private T whatIsThis;
    // use T as a method’s parameter type
    public void doThis(T input) { ... }
    // use T as a method’s return type
    public T doThat( ... ) {
        return new T( ... );
    }
}
```
Parameterized Classes

- Instantiating a parameterized class
  
  ```java
  Generic<String> g = new Generic<String>();
  ```

- Use methods with objects of the actual type
  
  ```java
  g.doThis("Hello");
  String s = g.doThat( ... );
  ```

- The compiler can verify the correctness of any parameters passed or assignments of the return values

- No casting of data types should be required (If it is, you aren’t using generics correctly)
Parameterized Classes

- Note: The letter used inside <> is a dummy and can be <T> like C++ or <E> like Sun
- I prefer to use T based on C++ popularity and that is also what our textbook uses
- Only reference types - not primitive types - can be used as a generic type

```java
Generic<String> g = ... // OK
Generic<int> g = ... // Compile error
```
Parameterized Classes

- Must use a known class - not dummy letters
  
```java
  Generic<T> g = new Generic<T>(); // error
```

- Unless in a generic class where T is defined
  
```java
  public class AnotherGenericClass<T>
  {
    ...
    ...
    Generic<T> g = new Generic<T>(); // OK
    ...
  }
```
Parameterized Classes

- Don’t omit an identified `<type>` in new code
  
  ```java
  Generic g = new Generic(); // legacy code?
  ```

- Compiler will give incompatible type errors without an explicit cast (narrowing)
  
  ```java
  String s = g.doThat( ... ); // error
  String s = (String) g.doThat( ... ); // OK
  ```

- Compiler will give unchecked warnings
  
  ```java
  g.doThis("Hello"); // warning
  ```
Parameterized Classes

• Can’t instantiate arrays of the generic data type without using a “trick”
  T [] t = new T[10]; // compile error
  T [] t = (T []) new Object[10]; // OK

• Can’t instantiate arrays of a parameterized class without using a slightly different “trick”
  ArrayList<String>[] a =
    (ArrayList<String>[]) new ArrayList[10];
  Just casting a new Object[10] compiles OK
  but throws an exception at run time (Ouch!)
Parameterized Classes

- When you use either of the above “tricks”, the compiler will give you an “unchecked” warning
- Normally, we would “fix” the code to get rid of the warning but here we can’t “fix” it
- Use the compiler SuppressWarnings directive
- Place this line ahead of the method header
  ```java
  @SuppressWarnings("unchecked")
  ```
- That directive will allow a “clean” compilation
Parameterized Classes

- Static members can’t use generics because there is only one copy of the code for all of the parameterized class objects instantiated with possibly different generic types

```java
public class BadExample<T> {
    private static T count;  // error
    public static T method() // error
        {...}
}
```
Parameterized Classes

• Don’t invoke static methods of parameterized classes with a generic type specified
  
  // the following is an error
  int n = BadExample<Integer>.method();
  // a static method must be invoked as
  int n = BadExample.method();

• But it is OK to instantiate the class with a generic to use its non-static methods
Customizing a Collection

• If we need a collection class that is similar to any library collection class, we can extend it
• The ArraySet class we need for Bingo Lab 3 is almost the same as the ArrayList class
• We can extend the ArrayList class without duplicating all of the code in ArrayList class
• We add attributes and override methods as needed
Customizing a Collection

• In Bingo Lab 3, we need a set collection to contain BingoBall objects for drawing numbers randomly
• You will extend the java.util.ArrayList<T> collection to create an ArraySet<T> collection
• It needs two features different from ArrayList<T>
  – Its add method does not allow duplicate objects to be added because no duplicate numbers should be drawn
  – It has a removeRandom method to draw balls
• A UML diagram for the Bingo application is shown in the next slide
UML Diagram for Bingo Application

- Bingo
  - main (args : String[]) : void

- ArraySet<BingoBall>
  - rand : Random
  - add(T element) : boolean
  - removeRandom() : T

- BingoBall

- java.util.ArrayList<T>

- <<interface>> Iterable<T>

Relationships:
- Bingo uses java.util.ArrayList<T>
- Bingo uses ArraySet<BingoBall>
- ArraySet<BingoBall> uses Bingo
- ArraySet<BingoBall> extends java.util.ArrayList<T>
- java.util.ArrayList<T> implements <<interface>> Iterable<T>
Customizing a Collection

• We write the code for an ArraySet<T> class:
  – Include an extends clause for ArrayList<T>
  – Include an attribute to hold a Random object
  – Write a constructor to instantiate Random object
  – Override the ArrayList add method to check for duplicates using parent’s contains method before adding an object (call the parent’s add method)
  – Add a removeRandom method that calculates a random integer and returns the return value from the parent’s remove method at that integer value
Inner Classes

- Sometimes when we write the code for a collection class, we need an “inner class” as a “helper” to aggregate data elements.
- The outer class is defined in a file with the outer class name plus .java as usual.
- The inner class is defined inside the { } of the public outer class.
- An “inner class” is usually declared private, but can be public if needed.
Inner Classes

• A public class with a private inner class

```java
public class OuterClass {

    // outer class attributes and methods

    private class InnerClass {

        // inner class attributes and methods
    }
}
```
Inner Classes

• Code outside the public outer class cannot instantiate an instance of a private inner class or use a reference to call its methods.

• Code in the public outer class’s methods can instantiate objects of an inner class and keep or use their references in its own data / code.

• Code in inner class methods can access the attributes or methods of the instance of the public class that instantiated the inner class.
Use of Exceptions

• In some cases, there may be constraints that prevent the execution of a method for a specific instance of a collection object.
• For example, a remove operation cannot be performed on a collection that is empty.
• The `remove` method for the class may be coded to check for an empty collection.
• If a collection is empty, `remove` may throw an `EmptyCollectionException`.
Use of Exceptions

- Hence, a method of any class may have a throws clause in its header, e.g. the class's remove method
  
  ```java
  T remove() throws EmptyCollectionException
  ```

- To throw an exception, the method uses:
  
  ```java
  throw new EmptyCollectionException("string");
  ```

- The exception itself is defined as a class:
  
  ```java
  public class EmptyCollectionException extends RuntimeException {
    public EmptyCollectionException (String coll) {
      super ("The " + coll + " is empty.");
    }
  }
  ```