CSC 172– Data Structures and Algorithms

Lecture 3

Spring 2018

TuTh 3:25 pm – 4:40 pm

Agenda

- Administrative aspects
- Java Generics

Chapter 1

ADMINISTRATIVE ASPECTS

Workshops

- Workshops
  - Workshops begin on this Sunday (January 28th)
  - Remember: Workshops run on Sun, Mon, Tues, Wed
  - One Session every week

Labs

- Labs
  - Already started
    - Runs on Mon, Tues, Wed, Thur
    - Two Sessions every week.

- Lab 2
  - Will be released on this Sunday (Sep 10)
  - No Lab demo required

Reading

- Java Generics (For the lab and next quiz)

- Chapter 3 and Chapter 4
  - From http://lti.cs.vt.edu/LTI_ruby/Books/CS172/html/

JAVA GENERICS

- Resources:
  - https://docs.oracle.com/javase/tutorial/java/generics/index.html
  - https://docs.oracle.com/javase/tutorial/extra/generics/index.html

Error or no error?

```
1 package lec3;
2
3 import java.util.*;
4
5 public class Gen1 {
6
7     public static void main(String[] args) {
8         List list = new ArrayList();
9         list.add("hello1");
.0         String s = list.get(0);
.1         System.out.println(s);
.2     }
.3
.4 }
.5
```

```
Exception in thread "main"
java.lang.Error: Unresolved
compilation problem:
    Type mismatch: cannot
convert from Object to String
```

Solution 1

```
1  package lec3;
2
3  import java.util.*;
4
5  public class Gen2 {
6
7      public static void main(String[] args) {
8          List list = new ArrayList();
9          list.add("hello2");
10         String s = (String) list.get(0);
11         System.out.println(s);
12     }
13
14 }
```

What about this?

```
public class Gen2a {

    public static void main(String[] args) {
        List list = new ArrayList();
        list.add(5);
        String s = (String) list.get(0);
        System.out.println(s);
    }

}
```

No compile time error

```
Exception in thread "main"
java.lang.ClassCastException:
java.lang.Integer cannot be cast
to java.lang.String
```

Solution 2

```
1  package lec3;
2
3  import java.util.*;
4
5  public class Gen3 {
6
7      public static void main(String[] args) {
8          List<String> list = new ArrayList<>();
9          list.add("hello3");
10         String s = list.get(0);    // no cast
11         System.out.println(s);
12     }
13
14 }
15
```

Solution 2a

```
public class Gen4 {

    public static void main(String[]
args) {
        List<String> list = new
ArrayList<>();
        list.add("Hello");
        int i = list.get(0);
        System.out.println(i);
    }


}
```

```
Exception in thread "main"
java.lang.Error: Unresolved
compilation problem:
    Type mismatch: cannot convert
from String to int
```

Solution 2b

```
public class Gen4b {

    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add(5);
        String i = list.get(0);
        System.out.println(i);
    }


}
```

```
Exception in thread "main"
java.lang.Error: Unresolved
compilation problem:
    The method add(int, String)
in the type List<String> is not
applicable for the arguments
(int)
```

Generics: Avoid or Embrace?

- Stronger type checks at compile time
  - A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety.
  - Fixing compile-time errors is easier than fixing runtime errors, which can be difficult to find.

- Elimination of casts

- Enabling programmers to implement generic algorithms
  - Programmers can implement generic algorithms
    - work on collections of different types
    - can be customized
    - are type safe
    - easier to read.

Another Example (A typical class)

```
public class Box {
    private Object object;

    public void set(Object object) { this.object = object;
}
    public Object get() { return object; }

    public static void main(String[] args) {
        Box box_int = new Box();
        Box box_String = new Box();

        box_int.set(5);
        int x = (int)box_int.get();
        System.out.println(x);

        box_String.set("Hello");
        String y = (String)box_String.get();
        System.out.println(y);


//      box_int.set("CSC172");
//      int x1 = (int) box_int.get();
//      System.out.println(x1);
    }
```

- What if we uncomment the last block

```
21          // Comment vs Uncomment
22          /*
23          box_int.set("CSC172");
24          int x1 = (int)box_int.get();
25          System.out.println(x1);
26          */
```

```
Exception in thread "main" Hello
java.lang.ClassCastException: java.lang.String cannot be cast to java.lang.Integer
        at lec3.Box.main(Box.java:23)
```

Solution: Generic Class

```
public class Box2<T> {
     // T stands for "Type"
     private T t;

     public void set(T t) { this.t = t; }
     public T get() { return t; }
     public static void main(String[] args) {
          Box2<Integer> box_int = new Box2<>();
          Box2<String> box_String = new
Box2<>();

          box_int.set(5);
          int x = box_int.get();
          System.out.println(x);

          box_String.set("Hello");
          String y = box_String.get();
          System.out.println(y);

          // Compilation Error
          /*
          box_int.set("CSC172");
          int x1 = (int)box_int.get();
          System.out.println(x1);
          */

     }

}
```

Generic Methods

- *Generic methods* are methods that introduce their own type parameters.

- Similar to declaring a generic type, but the type parameter's scope is limited to the method where it is declared.

- Static and non-static generic methods are allowed, as well as generic class constructors.

Example

```
public class Util {
    public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {
        return p1.getKey().equals(p2.getKey()) &&
               p1.getValue().equals(p2.getValue());
    }
}
```

Not a generic class. But The class contains a generic method

```
public class Pair<K, V> {

    private K key;
    private V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public void setKey(K key) { this.key = key; }
    public void setValue(V value) { this.value = value; }
    public K getKey()   { return key; }
    public V getValue() { return value; }
}
```

Generic Class

Example (cont.)

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = Util.<Integer, String>compare(p1, p2);
```

The type has been explicitly provided, as shown in bold. Generally, this can be left out and the compiler will infer the type that is needed:

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = Util.compare(p1, p2);
```

Coding Time

- Write a method which takes 2 arguments:
  - 1. An array of integer `anArray`
  - 2. An integer `elem`

  - Find how many integers in this array are greater than `elem`

- Write a main method to test your code

Method (Not Generic)

```
public static int countGreaterThan(int[] anArray, int elem)
{
    int total = 0;
    for (int val:anArray) {
        if (val > elem) {
            total++;
        }
    }
    return total;
}
```

Coding Time

- Now covert the method into a generic method which can handle any array.

- Your answer does not necessarily need to be correct (there are a few unknown we did not talk about)
- A good way to learn!

Method (Generic)

```
public static <T extends Comparable<T>> int
              countGreaterThan(T[] anArray, T elem)
{
    int total = 0;
    for (T val:anArray) {
        if (val.compareTo(elem) > 0) {
            total++;
        }
    }
    return total;
}
```

WILDCARDS

Method that Prints All Elements in a Collection

```
void printCollection(Collection<Object> c) {
    for (Object e : c) {
        System.out.println(e);
    }
}
```

only takes Collection<Object>
which is **not** a supertype of all kinds of collections!

What is the supertype of all kinds of collections?

- `Collection<?>`

OK

```
void printCollection(Collection<?> c) {
    for (Object e : c) {
        System.out.println(e);
    }
}
```

Error

```
Collection<?> c = new ArrayList<String>();
c.add(new Object()); // Compile time error
```

Bounded Wildcards

```
public abstract class Shape {
    public abstract void draw(Canvas c);
}

public class Circle extends Shape {
    private int x, y, radius;
    public void draw(Canvas c) {
        ...
    }
}

public class Rectangle extends Shape {
    private int x, y, width, height;
    public void draw(Canvas c) {
        ...
    }
}
```

Bounded Wildcards

These classes can be drawn on a canvas:

```
public class Canvas {
    public void draw(Shape s) {
        s.draw(this);
    }
}
```

Draw them all

```
public void drawAll(List<Shape> shapes) {
    for (Shape s: shapes) {
        s.draw(this);
    }
}
```

drawAll() can only be called on lists of exactly Shape
it cannot, for instance, be called on a List<Circle>

Solution

```
public void drawAll(List<? extends Shape> shapes) {
    ...
}
```

List<? **extends** Shape> is an example of a *bounded wildcard*. we know that this unknown type is in fact a subtype of Shape.

Still!

```
public void addRectangle(List<? extends Shape> shapes) {
    // Compile-time error!
    shapes.add(0, new Rectangle());
}
```

Solution: super

```
public void addRectangle(List<? super Rectangle> shapes)
{
    shapes.add(0, new Rectangle());
}
```

super (lower bound)

```
public void addRectangle(List<? super Rectangle> shapes)
{
    shapes.add(0, new Rectangle());
}
```

Another Solution: Generic Method

```
public <T extends Shape> void addRectangle(List<T> shapes)
{
    shapes.add(0, new T());
}
```

Which one to choose?

- Generic Method vs Wildcard

```
interface Collection<E> {
    public boolean containsAll(Collection<?> c);
    public boolean addAll(Collection<? extends E> c);
}
```

```
interface Collection<E> {
    public <T> boolean containsAll(Collection<T> c);
    public <T extends E> boolean addAll(Collection<T> c);
    // Hey, type variables can have bounds too!
}
```

Generic Method vs Wildcard

```
class Collections {
    public static <T> void copy(List<T> dest, List<? extends T> src) {
    ...
}
```

VS

```
class Collections {
    public static <T, S extends T> void copy(List<T> dest, List<S> src) {
    ...
}
```

Using wildcards is clearer and more concise than declaring explicit type parameters , and should therefore be preferred whenever possible.

A great explanation

- https://stackoverflow.com/a/4343547
- Remember *PECS*: **"Producer Extends, Consumer Super"**.
-

extends (stack overflow)

The wildcard declaration of `List<? extends Number> foo3` means that any of these are legal assignments:

```
List<? extends Number> foo3 = new ArrayList<Number>();  // Number "extends" Number (
List<? extends Number> foo3 = new ArrayList<Integer>(); // Integer extends Number
List<? extends Number> foo3 = new ArrayList<Double>();  // Double extends Number
```

1. **Reading** - Given the above possible assignments, what type of object are you guarenteed to read from `List foo3`:
   - You can read a `Number` because any of the lists that could be assigned to `foo3` contain a `Number` or a subclass of `Number`.
   - You can't read an `Integer` because `foo3` could be pointing at a `List<Double>`.
   - You can't read a `Double` because `foo3` could be pointing at a `List<Integer>`.
2. **Writing** - Given the above possible assignments, what type of object could you add to `List foo3` that would be legal for **all** the above possible `ArrayList` assignments:
   - You can't add an `Integer` because `foo3` could be pointing at a `List<Double>`.
   - You can't add a `Double` because `foo3` could be pointing at a `List<Integer>`.
   - You can't add a `Number` because `foo3` could be pointing at a `List<Integer>`.

You can't add any object to `List<? extends T>` because you can't guarantee what kind of `List` it is really pointing to, so you can't guarantee that the object is allowed in that `List`. The only "guarantee" is that you can only read from it and you'll get a `T` or subclass of `T`.

super (stack overflow)

The wildcard declaration of `List<? super Integer> foo3` means that any of these are legal assignments:

```
List<? super Integer> foo3 = new ArrayList<Integer>();  // Integer is a "superclass"
List<? super Integer> foo3 = new ArrayList<Number>();   // Number is a superclass of
List<? super Integer> foo3 = new ArrayList<Object>();   // Object is a superclass of
```

1. **Reading** - Given the above possible assignments, what type of object are you guaranteed to receive when you read from `List foo3`:
   - You aren't guaranteed an `Integer` because `foo3` could be pointing at a `List<Number>` or `List<Object>`.
   - You aren't guaranteed an `Number` because `foo3` could be pointing at a `List<Object>`.
   - The **only** guarantee is that you will get an instance of an `Object` or subclass of `Object` (but you don't know what subclass).
2. **Writing** - Given the above possible assignments, what type of object could you add to `List foo3` that would be legal for **all** the above possible `ArrayList` assignments:
   - You can add an `Integer` because an `Integer` is allowed in any of above lists.
   - You can add an instance of a subclass of `Integer` because an instance of a subclass of `Integer` is allowed in any of the above lists.
   - You can't add a `Double` because `foo3` could be pointing at a `ArrayList<Integer>`.
   - You can't add a `Number` because `foo3` could be pointing at a `ArrayList<Integer>`.
   - You can't add a `Object` because `foo3` could be pointing at a `ArrayList<Integer>`.

Functional Programming with Java 8

```
import java.util.function.*;

public class FuncObject {
  public static void main(String[] args) {

Function<Integer,Integer> add1 = x -> x + 1;
Function<String,String> concat = x -> "Hello, "  +  x;


Integer six= add1.apply(5);
String answer = concat.apply("Tom");

System.out.println(six);
System.out.println(answer);


  }
}
```

- Link: https://dzone.com/articles/functional-programming-java-8

Acknowledgement

- ORACLE Java Documentation