CPSC 324
Topics in Java Programming

Lecture 21

Today …
• Project Addendum
• More on Generics
• Finish up exercises
• Reading Assignment
  – Core: Ch. 12 pp. 625-640 (more on generics)
**Group Project Addendum**

You must create javadocs for your application ...

- This means running the `javadoc` command
- Placing the `.html` files created under a subfolder in your `public_html` directory on ada
- Putting a link to the corresponding URL in your project report

---

**Some Notes on Generics**

"Unchecked Warnings"

```java
List<Integer> list = new ArrayList();
```

- Results in a warning ... `Object` → `Integer`
- In this case, must add type parameters to “both sides” of `=`’s
- Often happens with legacy (i.e., non generic) classes
Some Notes on Generics

In addition to casts, generics provide **type checking**

```java
List<Integer> list = new ArrayList<Integer>();
list.add("Hello World!"); // a compile error!
list.add(1); // this type checks via auto boxing
```

Some Notes on Generics

Type conversion (failure)

```java
List<Integer> ints = Arrays.asList(1, 2, 3);
// compile (type) errors:
List<Number> nums = ints;
List<Number> nums = (List<Integer>)ints;
But should these be errors?
```

List<Number> seems like a supertype of List<Integer>

- There are some issues though ...
  ```java
  nums.add(3.14); // ints now contains a double!
  ```
- Handling this subtyping requires more machinery ...
**Generic Methods**

- So far we have used *generic classes*
  - Pair and MyLinkedList
  - e.g., by adding `<T>` or `<T1,T2>` to the class definition ...
    ```java
    public class Pair<T1,T2> { ... }
    public class MyLinkedList<T> { ... }
    ```

- Sometimes it’s also convenient to have *generic methods*
  - That is, pass type parameters into method calls

```java
public class Lists {
    public static <T> List<T> listify(T obj) {
        List<T> list = new ArrayList<T>();
        list.add(obj);
        return list;
    }
}
```

- Calling the method (e.g., in main of Lists):
  ```java
  List<String> myList = listify("Hello World!");
  System.out.println(myList);  // ["Hello World!"]
  ```
Generic Methods

• Can also be called by passing in the type explicitly

    List<String> myList = Lists.<String>listify(“Hello World!”);

Note position of type param

    List<String> myList = <String>listify(“Hello World!”);

This won’t work!!! (Compiler gets confused)

• Not needed in this (or most) cases
  – There are some cases where the compiler needs more information about types
  – Then we do need to explicitly give the type parameter
  – We’ll see an example soon

Generic Methods

• Consider this version of listify

    public static <T> List<T> listify(T[] objs) {
        List<T> list = new ArrayList<T>();
        for(T obj : objs)
            list.add(obj);
        return list;
    }

• How we might use it

    List<String> l1 = Lists.listify(new String[] {“a”, “b”});
    assert l1.get(0).equals(“a”);
    List<Integer> l2 = Lists.listify(new Integer[] {1, 2, 3});
    assert l2.get(0) == 1;  // boxing/unboxing is cool!
Varargs

• The `vararg` feature is “syntactic sugar” for array args
  – Avoids creation of static arrays in cases like our example

• Here is an even fancier version of `listify`
  
  ```java
  public static <T> List<T> listify(T... objs) {
      List<T> list = new ArrayList<T>();
      for(T obj : objs)
          list.add(obj);
      return list;
  }
  
  Much more convenient and concise
  
  And calls
  
  List<String> l2 = Lists.listify("a", "b");
  List<Integer> l3 = Lists.listify(1, 2, 3);
  ```

Varargs

• Behind the scenes:
  
  ```java
  public static <T> List<T> listify(T... objs) {
      List<T> list = new ArrayList<T>();
      for(T obj : objs)
          list.add(obj);
      return list;
  }
  
  Equivalent to: T[] objs (an array of type T)
  
  So in `listify` we can access `objs` as if it is an array ...
  
  ```java
  for(int i = 0; i < objs.length; i++)
      list.add(objs[i]);
  ```
Using Varargs

• Recall the new signature for listify
  
  public static <T> List<T> listify(T... objs) {...}

• What do you think happens in this call?
  
  List<Integer> list = Lists.listify();
  – We aren’t passing in any type information
  – So effectively only List<Object> can be inferred
  – This results in an unchecked warning !!!

• Should pass in the type parameter explicitly:
  
  Lists<Integer> list = Lists.<Integer>listify();

Using Varargs

• Varargs can be used with other method arguments
  – But only one vararg can occur and it must be the last argument
  
  public static <T> void addAll(List<T> list, T... ts) {
      for(T t : ts) list.add(t);
  }
  – To call:
  
  List<Integer> aList = Lists.<Integer>listify();
  addAll(aList, 1, 2, 3);
  System.out.println(aList); // [1, 2, 3]

• Still not widely used by Java programmers
  – E.g., overriding and generics can be issues (unchecked exceptions)
Wildcards with Extends

• Consider this example again ...

```java
List<Integer> ints = Arrays.asList(1, 2);
List<Number> nums = ints;
nums.add(3.14);
assert ints.toString().equals("[1, 2, 3.14]"tm);```

**What is the problem here?**

– ints is of type List<Integer>
– And we added a double value to ints
– This could result in a *runtime type error* (youch!)
– No self-respecting type system has runtime type errors!!!

Wildcards with Extends

• Consider this example again ...

```java
List<Integer> ints = Arrays.asList(1, 2);
List<? extends Number> nums = ints;
nums.add(3.14);
assert ints.toString().equals("[1, 2, 3.14]"tm);
```

**What is going on here?**

– "? extends Number" means nums is a list of a T for T a subclass of Number
– The ? is called a "wildcard"
– In this example, we are only allowing the conversion
Wildcards with Extends

- Using wildcards with extends helps for “setters”
  
  ```java
  interface List<E> {
      public boolean addAll(List<? Extends E> c);
      ...
  }
  ```
  
- For example
  
  ```java
  List<Number> nums = new ArrayList<Number>();
  List<Integer> ints = Arrays.asList(1, 2, 3);
  nums.addAll(ints);
  // Without the wildcard this would give an unchecked warning!
  ```

Wildcards with Super

- Back to our previous example
  
  ```java
  List<Integer> ints = Arrays.asList(1, 2, 3);
  List<? extends Number> nums = ints;
  for(Number n : nums) System.out.println(n);
  ```
  
- Using wildcards with extends helps us get values
  - But not put values
  - For this we use wildcards with super …
Wildcards with Super

- Another example

```java
public static <T> void copy(List<? super T> dst, List<? extends T> src)
{
    for(int i = i < src.size(); i++)
        dst.set(I, src.get(i));
}
```

- Calling:

```java
List<Number> nums = Arrays.<Number>asList(4, 5, 3.14);
List<Integer> ints = Arrays.asList(1, 2);
Lists.copy(nums, ints)
assert nums.toString().equals("[1, 2, 3.14]");
```

- Finds a matching type for T ... i.e., Number or Integer

The Get and Put Principle

- Use an `extends wildcard` when you only get values out of a structure

- Use a `super wildcard` when you only put values into a structure

- Don’t use a wildcard if you both get and put

- In this case:

```java
copy(List<? super T> dst, List<? extends T> src)
```

- We only get values from src
- And we only put values into dst
More Information …

• Using generics is fairly straightforward
  – And can really *simplify* coding in Java (with foreach loops, etc)

• Defining (good) generic classes is a lot harder!
  – There are many factors to consider
  – Requires a lot of (advanced) Java understanding
  – A good book for learning more:

Next Time …

• The Java Collections Framework

• Start of “Potpourri”
  – Creating and using your own Exception types
  – Java packages
  – Creating and deploying Java applications
  – Reflection API and Threads (if time)