CMPE212 – Reminders

• Assn 4 is posted. Due a week from this Friday.
• Assn 3 sample solution is posted.
• Quiz 2 grading not underway...

Today

• Generics, Cont.
• Discuss a "Factory" method – the Emitter class in Assn 4.
• Start Lambda Functions (if we have time).

Last Time: Your Own Generic Methods

• From last lecture:

   public static <T> T getMidpoint(T[] a) {
     return a[a.length/2];
   }

   To invoke:

   String mid = Utility.getMidpoint(b);
   // b is some array of String's

Last Time: Your Own Generic Methods, Cont.

• How else could you write getMidpoint?

   public static Object getOMidpoint(Object[] objArray) {
     return objArray[objArray.length / 2];
   } // end getOMidpoint

   See Utility.java and TestUtility.java.

   Are generic methods better?

Why Generic Methods?

• In the example (strings is of type String[]):

   String mid = (String)Utility.getOMidpoint(strings);

   Suppose you wrote the following by mistake:

   Integer midI = (Integer)Utility.getOMidpoint(strings);

   You will not get a compilation error, but a "ClassCastException" will crash your program!

Why Generic Methods?, Cont.

• On the other hand:

   Integer midI = Utility.<String>getMidpoint(b);

   Gives you a compilation error and will not run.

   The compiler is enforcing type safety, which it cannot do when you are just using Objects.

   This advantage applies to any generic method or class.
Generic Methods and Java ≥ 7

- Constructors can also be generic. In a non-generic class you will need to use the <> in the header of the constructor.
- And you can use type inference as demonstrated in TestUtility.java:
  \[ \text{midStrings} = \text{Utility.getMidpoint(strings);} \]
- Same as:
  \[ \text{midStrings} = \text{Utility.<String>getMidpoint(strings);} \]

Generics & Backwards Compatibility

- Generic code can make for very versatile programs that can be applied to a range of objects – saving you a great deal of boring coding work.
- But, generic code will not work easily with pre-Java 5.0 code.
- And, of course, the Java 7 type inference stunts will not work with pre-Java 7.0 code.
- Generics are a bit like Templates in C++ and C#, but are implemented in a different way.

Generics in Modern Java

- This discussion assumes newer versions of Java (≥ 1.7)
- Generics allow you to write powerful, compact code that can work with many object types. Particularly useful with hierarchies.
- Type inference, bounding, wildcards and the use of the Class<T> object adds power and convenience.
- Let's experiment!

Wildcards Demo

- TestWildcards.java
- Uses the Person hierarchy from last week.
- Plays with wildcards, bounded generic methods, and how to create an array of generic classes.

Summary: Wildcards

- If Child is a subclass of Parent then this works:
  \[ \text{Parent obj = new Child();} \]
- But this will not work:
  \[ \text{ArrayList<Parent> alObj = ArrayList<Child>();} \]
- This will work:
  \[ \text{ArrayList<T> alObj = ArrayList<Child>();} \]
- A "wildcard" is the ? within the < >.
- A wildcard can provide a super type generic class for other generic classes.
Summary: Wildcards, Cont.

- However, a generic type with a wildcard is not mutable.

- You can get around this using generic methods without the wildcard. But:
  - This is more restrictive and potentially "brittle".
  - Avoid casting using the generic type (using `T`). This may lead to "Unchecked Casts" which reduces the type safety of the generic method. (Hint: use `@SuppressWarnings("unchecked")`)
  - Use of `Class<T>` may be a way around this problem.

ArrayList<?>[] is the only possible type that allows the creation of an array of generic classes.

- But, why would you want to?
- Easier to create an ArrayList of ArrayLists, for example.

Aside – Lower Bound for Wildcard

- In `? extends T` the extends keyword represents an upper bound for the type – `T` or any sub-type of `T`.

- You can also use super as in `? super T` which provides a lower bound for the type – `T` or any super-type of `T`.

Emitters in Assignment 4

- The Emitter class follows the "Factory" pattern.
- Instead of having one Emitter class for each Particle type, how can you have just one Emitter class that accepts the type of Particle to be generated?
- Possible approaches:
  - Make Emitter a generic class and supply the Particle type for `T`.
  - Make the launch method generic.
  - Supply an empty ArrayList<? extends Particle> argument to launch.
  - Supply a Class<? extends Particle> argument to the Emitter constructor.
  - Supply a Class<? extends Particle> argument to the launch method.
  - Have a static method in a Particle that returns an instance of the Particle.

Emitters in Assignment 4 - Restrictions

- If `T` is a generic type:
  - You cannot instantiate an object of type `T`.
  - Type `T` does not have a `.class` attribute.

- If you use a wildcard in a type:
  - An object (including an ArrayList<T>) created with a wildcard is immutable.

- If you have a `Class<T>` object you must use Reflection to create an instance of the Particle.
  - You must either use the same constructor parameter list for each child class of Particle or use just the default constructor and mutators on the instance.
Anonymous Class Example - Cont.

- `MessageSender` is an interface, not an Object:

  ```java
  public interface MessageSender {
      void sendGreeting(String name);
  }
  ```

- Java ≥ 8 now has a very tidy solution to using messy anonymous classes – Lambda Functions:

```java
public class LambdaDemo {
    public static void main(String[] args) {
        MessageSender ms = name -> System.out.println("Hello " + name);
        ms.sendGreeting("Steve");
    }
}
```

The Lambda Version

```java
public class LambdaDemoAgain {
    interface MessageSender {
        void sendGreeting(String aName);
    }
    public static void main(String[] args) {
        MessageSender ms = name -> System.out.println("Hello " + name);
        ms.sendGreeting("Steve");
    }
}
```

A Lambda Function

- Kind of like an "anonymous method".
- Syntax:
  - Parameters for the method first. No parameters, use `{ }`, one parameter by itself – no brackets required, more than one use `(a, b, etc.)`. No types required.
  - Then the fancy -> "arrow".
  - Then the method body. Put more than one statement inside `{ }`.
- Could even define an inner interface:

```
public class LambdaDemoAgain {
    interface MessageSender {
        void sendGreeting(String aName);
    }
    public static void main(String[] args) {
        MessageSender ms = name -> System.out.println("Hello " + name);
        ms.sendGreeting("Steve");
    }
}
```

Lambda Functions, Cont.

- Note how the abstract method in the interface determines the structure of the lambda function – the parameter and parameter type, the method name and the lack of a return statement.
- These functions could be useful! *(Especially when attaching events to GUI components.)*
- Only certain interface structures can be used.

```java
Suppose you want to have a method that only displays certain members of a collection, depending on a criteria that is specified outside the method and provided as an argument.

- You don’t want to hard code the criteria inside the display method.
- First technique: Supply an object implementing an interface that has a "filter" method that returns a true or false. See TestFilter1.java
```
Lambda Functions, Cont.

- Second technique: TestFilter2.java – use an anonymous class instead.
- This is the best version, so far!

But how does it work?

- The structure of the method implemented by the lambda function is specified by the interface type used in the displaySome method.
- The signature is:
  ```java
  boolean check(Person)
  ```
- The compiler knows from this that the type to the left of the `->` must be a Person and the expression to the right of the `->` must evaluate to a boolean.

Filter is an example of a Functional Interface.

- These interfaces can only contain a single abstract method.
- Lambdas can only be created using Functional Interfaces.
- You can use the `@FunctionalInterface` annotation to make sure your interface is OK.

Turns out the java.util.function package has many pre-defined generic functional interfaces.

- See the API, `svp`.
- The one that matches our check function signature is called `Predicate<T>`. It has the abstract method signature:
  ```java
  boolean test(T)
  ```
- See TestFilter4.java.
- Even better than the last best one!