CMPE212 – Reminders

• Assn 4 not posted yet. Still…
• Assn 3 sample solution is posted.

Today

• Inheritance, Cont.:
  – Another, better version of the OOP Zilch game that eliminates duplicated code.
• Generics
  – Start by looking at an existing Generic Class – ArrayList<T>
  – How to build a Generic Class.

Fixing Up the Zilch OOP Version

• From before: One problem with the last OOP version was that there was quite a bit of duplicated code in the classes AIPlayer and HumanPlayer.
• Let's have a quick look at the previous OOP version.
• Now, this duplicated code can be put in a parent class: Player, and then AIPlayer and HumanPlayer can extend this parent class. Let's have a look. Note that Player is abstract – why does this have to be the case?
• No other classes have changed.
• A much better structure!

What's Next?

• You have everything you need to work on Assignment 4.
• You are also ready to do Exercise 9.
• Next topic is Generics in Java:
  – Start by looking at a Generic Collection type from the API – ArrayList<T>.
  – This collection has been used in demo and assignment testing code already.

The ArrayList<T> Class

• This is a generic data structure class.
• The ArrayList<T> class resides in the java.util package. Stores and returns Objects of type T.
• (Referred to as "ArrayList<E>" in the API – same difference…)
• You can use the class without specifying a size.

ArrayList<T> Class, Cont.

• In newer versions of Java you do not have to specify the type twice. So, you can do:
  ArrayList<T> list_name = new ArrayList<>();
• For example:
  ArrayList<Double> myList = new ArrayList<>();
• The empty <> is called the “diamond”...
• This is an aspect of what is called "type inference".
ArrayList Class<T>, Cont.

- To add elements to the collection, for example:
  
  ```java
  myList.add(Double.valueOf(456.78))  
  myList.add(456.78);  // Automatic Boxing
  ```

- You can also use `addAll(collection)` to add a bunch of elements at once.
- To get the size of the collection, invoke the `size()` method:
  ```java
  int myListSize = myList.size();
  ```

ArrayList Class<T>, Cont.

- The `get(index)` method returns the element of type T at the given index. Note that index positions are numbered from zero (of course!). ([] are not used!!)
- The `set(position, new_value)` method changes the element at the given position.
- To insert an element, invoke `add(position, new_value)`. All elements are moved up, and the new value is added at the given position.
- The method, `remove(position)` removes the element at the provided position. Also have `removeAll(collection)`

ArrayList Class<T>, Cont.

- If you are done adding elements to the ArrayList, invoke `trimToSize()` to remove empty element locations.
- Also, `clone()` does not work properly because it does not return an independent, un-aliased copy. The objects in the collection will still be aliased.

ArrayList Class<T>, Cont.

- `toArray()` returns the underlying array of type T[] as Object[].
- `toArray(tArray)`:
  - Supply `tArray` which is an instantiated array of type T. The size will be increased if necessary, so just use a size of zero.
  - For example, if T is String then invoke as in:
    ```java
    arrayList.toArray(new String[0])
    ```
  - This method returns the array String[] of the exact size needed to hold all the elements in the `arrayList`.

ArrayList Class<T>, Cont.

- Other methods:
  - `contains()`, `containsAll()`, `indexOf()`, `removeIf(Predicate)`, `removeRange()`, `subList()`, `sort()`

  And, `equals()` is implemented in such a way that not only must the two collections be of the same List type and size, each pair of elements must also be equal. (So equals will have to be implemented properly for the element type as well).

ArrayList Class<T>, Cont.

- Once you have accessed an element of an `ArrayList<T>`, you do not have to cast it back to type T, it is already of the correct type.
- For example, to get the number back out of the first element in `myList`:
  ```java
  double aVal = myList.get(0);
  ```
- Also uses automatic unboxing.
- This is an advantage over using a collection like `Object[]`, for example where you would have to cast the returned `Object`. 
Array List Class\textless{}T\textgreater{}, Cont.

- Final note:
- What is the underlying data structure for an ArrayList and how do you find out?
- Do you use the ArrayList\textless{}T\textgreater{} class for speed or for convenience?

Your Own Generic Classes

- Also called "Parameterized Classes" - kind of like having a parameter in the class header - except it only specifies a type and does not pass a reference, like a parameter does.
- See the next slide for an example:

Your Own Generic Classes, Example

```java
public class Sample\textless{}T\textgreater{} {
    private T data;

    public void setData(T newData) {
        data = newData.clone();
    }

    public T getData () {
        return data;
    }
}
```

Your Own Generic Classes, Example - Cont.

- Using the Sample generic class:

```java
Sample\textless{}Integer\textgreater{} num1 = new Sample\textless{}Integer\textgreater{}();
num1.setData(Integer.valueOf(45));
System.out.println(num1.getData().intValue());
```

- Prints out 45.

Your Own Generic Classes, Example - Cont.

- Any object type can be used for "T":

```java
Sample\textless{}String\textgreater{} string1 = new Sample\textless{}String\textgreater{}();
string1.setData("Hello!");
System.out.println(string1.getData());
```

- Prints out Hello!.
Generic Classes, Limitations for T

• You can use an array type, so "<int[]>" would be legal.
• You cannot specify primitive types, so "<int>" would not be legal.
• Neither can you use Exception classes - (Why would you want to?)
• You can use interfaces and abstract types.

Your Own Generic Classes, Constructor

• A constructor for our Sample<T> class:
  public Sample (T newData) {
    data = newData.clone();
  }

  • Note that there is no use of <T> in the constructor definition.

Your Own Generic Classes, Constructor, Cont.

• Using the constructor and automatic boxing:
  Sample<Double> num3 = new Sample<>(56.7);
  System.out.println(num3.getData());

Your Own Generic Classes, Notes

• You can specify any number of type parameters:
  public class Sample2<T1, T2, T3> {...

Your Own Generic Classes, Bounding

• You can also "bound" the parameters by specifying a root class:
  public class Sample3<T extends RootClass> {
    ...
  }

  • So, only classes that extend RootClass, or RootClass itself, can be used for T, in this example.
  • You can specify an upper bound for multiple classes and/or interfaces:
    public class Sample3<T extends A & B & C> {
      ...
    }

Your Own Generic Classes, Bounding

• The extends keyword allows you to specify an "upper" bound.
  • You could use the "super" keyword to specify a "lower" bound:
    public class Sample3<T super Integer> {
      ...
    }

    • In this case T could be Integer, Number or Object.
Your Own Generic Classes, Upper Bounding

- Why bother?
- Upper bounding is like implementing an interface – it will ensure that the object T will have the methods in the class you are extending.
- For example `<T extends Comparable<T>>` will ensure that the type used for T will have a .compareTo() method.

Your Own Generic Methods

- These can be defined in any class, including non-Generic classes.
- For example in the class Utility:

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

- To invoke:

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

Your Own Generic Methods, Cont.

- The `<T>` has to come before any use of T as a type in the method header.
- When invoking a generic method in newer Java versions you can ignore the `<...>` in the invocation:

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

- This is another example of 'type inference'. The type for T is inferred by looking at the type used in the expression.

Why Generic Methods?

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

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

- Suppose you wrote the following by mistake:

  ```java
  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:

  ```java
  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:
  ```java
  midStrings = Utility.getMidpoint(strings);
  ```
  ```java
  midStrings = 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.