Topic 6
Generic Type Parameters

"Get your data structures correct first, and the rest of the program will write itself."
- *David Jones*

Back to our Array Based List

- Started with a list of ints
- Don't want to have to write a new list class for every data type we want to store in lists
- Moved to an array of `Object`s to store the elements of the list

```
// from array based list
private Object[] con;
```

Using Object

- In Java, all classes inherit from exactly one other class except Object which is at the top of the class hierarchy
  – therefore all classes are descendants of Object
- object variables can refer to objects of their declared type and any descendants
  – polymorphism
- Thus, if the internal storage container is of type Object it can hold anything
  – primitives handled by *wrapping* them in objects. int – Integer, char - Character

Difficulties with Object

- *Creating* generic data structures using the Object data type and polymorphism is relatively straight forward
- Using these generic data structrues leads to some difficulties
  - Casting
  - Type checking
- Code examples on the following slides

Clicker 1

- What is output by the following code?

```
GenericList list = new GenericList(); // 1
Street s = new Street("Boardwalk", 400,
                                    Color.BLUE);
list.add(s); // 2
System.out.print(list.get(0).getPrice());// 3
```

A. 400

B. No output due to syntax error at line // 1

C. No output due to syntax error at line // 2

D. No output due to syntax error at line // 3

E. No output due to runtime error.

Code Example - Casting

- Assume a list class

```
GenericList li = new GenericList();
li.add("Hi");
System.out.println(li.get(0).charAt(0));
// previous line has syntax error
// return type of get is Object
// Object does not have a charAt method
// compiler relies on declared type
System.out.println(
      ((String) li.get(0)).charAt(0) );
// must cast to a String
```

Code Example – type checking

```
//pre: all elements of li are Monopoly Properties
public void printPrices(GenericList li) {
     for (int i = 0; i < li.size(); i++) {
          Property temp = (Property) li.get(i);
          System.out.println(temp.getPrice());
     }
}
// what happens if pre condition not met?
```

"Fixing" the Method

```
//pre: all elements of li are Monopoly Properties

public void printPrices(GenericList li) {
     for(int i = 0; i < li.size(); i++) {
          // GACK!!!!
          if (li.get(i) instanceof Property) {
            Property temp = (Property) li.get(i);
            System.out.println(temp.getPrice());
          }
     }
}
```

Clicker 2 - Too Generic?

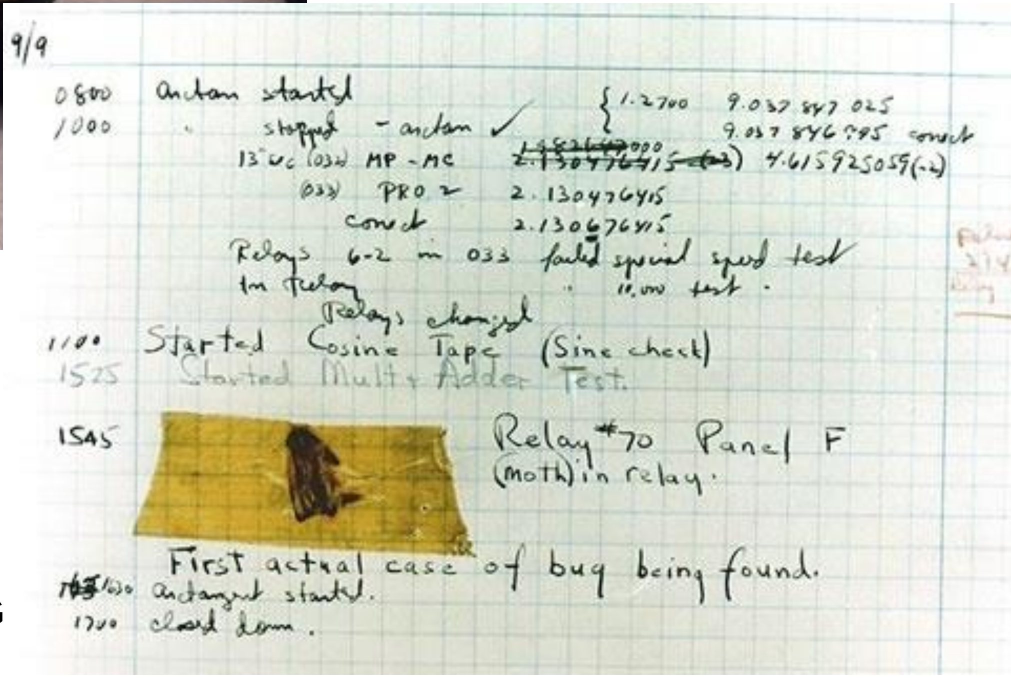
- Does this code compile?

```
GenericList list = new GenericList();
list.add("Olivia");
list.add(Integer.valueOf(12));
list.add(12); // autobox aka autowrap
list.add(new Rectangle(1, 2, 3, 4));
list.add(new GenericList());
```

A. No

B. Yes

Is this a bug or a feature?



9/9

0800 Arctan started

1000 " stopped - arctan ✓ {1.2700 9.037 847 025

9.037 846 795 correct

13" UC (032) MP - MC ~~1.982147000~~ ~~2.130476415~~ (-3)) 4.615925059(-2)

(033) PRO 2 2.130476415

correct 2.130676415

Relays 6-2 in 033 failed special speed test

In relay " 11,000 test.

Relays changed

1100 Started Cosine Tape (Sine check)

1525 Started Mult + Adder Test.

1545 Relay #70 Panel F

(moth) in relay.

First actual case of bug being found.

1630 Arctangent started.

1700 closed down.

Generic Types

- Java has syntax for *parameterized data types*
- Referred to as *Generic Types* in most of the literature
- A traditional parameter *has* a data type and can store various values just like a variable

```
public void foo(int x)
```

- Generic Types are ***like*** parameters, but the data type for the parameter is *data type*
  - like a variable that stores a data type
  - **<u>this is an abstraction</u>**. Actually, all data type info is erased at compile time and replaced with casts and, typically, variables of type Object

Making our Array List Generic

- Data type variables declared in class header

```
public class GenericList<E> {
```

- The `<E>` is the declaration of a data type parameter for the class
  - any legal identifier: `Foo`, `AnyType`, `Element`, `DataTypeThisListStores`
  - Java style guide recommends terse identifiers
- The value E stores will be filled in whenever a programmer creates a new `GenericList`

```
 GenericList<String> li =
                 new GenericList<>();
```

Modifications to GenericList

- instance variable

```
private E[] myCon;
```

- Parameters on
  – add, insert, remove, insertAll
- Return type on
  – get
- Changes to creation of internal storage container

```
myCon = (E[]) new Object[DEFAULT_SIZE];
```

- Constructor header does not change

Modifications to GenericList

- Careful with the equals method
- Recall type information is actually erased at compile time.
  - At runtime not sure what data type of elements are. (Unless we get into reflection.)
- use of wildcard
- rely on the elements equals methods

Using Generic Types

- Back to Java's ArrayList

```
ArrayList list1 = new ArrayList();
```

  - still allowed, a "raw" ArrayList
  - works just like our first pass at GenericList
  - casting, lack of type safety

Using Generic Types

```
ArrayList<String> list2 =
                 new ArrayList<String>();
```

– for list2 `E` stores `String`

```
list2.add( "Isabelle" );
System.out.println(
    list2.get(0).charAt(2) ); //ok
list2.add( new Rectangle() );
// syntax error
```

Parameters and Generic Types

- Old version

```
//pre: all elements of li are Strings
public void printFirstChar(ArrayList li){
```

- New version

```
//pre: none
public void printFirstChar(ArrayList<String> li){
```

- Elsewhere

```
ArrayList<String> list3 = new ArrayList<String>();
printFirstChar( list3 ); // ok
ArrayList<Integer> list4 = new ArrayList<Integer>();
printFirstChar( list4 ); // syntax error
```

Generic Types and Subclasses

```
ArrayList<Shape> list5 =
              new ArrayList<Shape>();
list5.add(new Rectangle());
list5.add(new Square());
list5.add(new Circle());
// all okay
```

- `list5` can store `Shape` objects and any descendants of `Shape`