Polymorphism

Outline

- Explanation of polymorphism.
- Using polymorphism to make generic data structures.

Polymorphism

- Another feature of OOP
  - literally “having many forms”
- Object variables in Java are polymorphic
- Object variables can refer to:
  - objects or their declared type

    AND
  - any objects that are descendants of the declared type

```
ClosedShape s;  // abstract class
s = new ClosedShape(); //illegal!
s = new Rectangle();//sub-class legal!
```

Data Type

- object variables have:
  - a *declared type*: Also called the static type.
  - a *dynamic type*. What is the actual type of the pointee at run time or when a particular statement is executed?
- Method calls are syntactically legal if the method is in the declared type *or any ancestor* of the declared type
- **The actual method that is executed at runtime is based on the dynamic type**
  - dynamic dispatch

Question

Consider the following class declarations:

```
public class BoardSpace
public class Property extends BoardSpace
public class Street extends Property
public class Railroad extends Property
```

Which of the following statements would cause a syntax error? Assume all classes have a default constructor.

A. `Object obj = new Railroad();`

B. `Street s = new BoardSpace();`

C. `BoardSpace b = new Street();`

D. `Railroad r = new Street();`

E. More than one of these

Question

Consider the following class declarations:

```
public class BoardSpace
public class Property extends BoardSpace
public class Street extends Property
public class Railroad extends Property
```

Which of the following statements would cause a syntax error? Assume all classes have a default constructor.

A. `Object obj = new Railroad();`

B. `Street s = new BoardSpace();`

C. `BoardSpace b = new Street();`

D. `Railroad r = new Street();`

E. More than one of these

What's the Output?

```
ClosedShape s = new Rectangle(1,2);
System.out.println( s.toString() );

s = new Rectangle(2, 3, 4, 5);
System.out.println( s.toString() );

s = new Circle(4, 5, 10);
System.out.println( s.toString() );

s = new ClosedShape();
System.out.println( s.toString() );
```

What's the Output?

```
ClosedShape s = new Rectangle(1,2);
System.out.println( s.toString() );
```

x: 1 y: 2

```
s = new Rectangle(2, 3, 4, 5);
System.out.println( s.toString() );
```

x: 2 y: 3

width: 4 height: 5

```
s = new Circle(4, 5, 10);
System.out.println( s.toString() );
```

x: 4 y: 5

radius: 10

```
s = new ClosedShape();
System.out.println( s.toString() );
```

Error: won't compile

Method LookUp

- To determine if a method is legal, the compiler looks in the class based on the declared type
  - if it finds it, great
  - if not, go to the super class and look there
  - continue until the method is found, or the Object class is reached and the method was never found. (Compile error)
- To determine which method is actually executed the run time system
  - starts with the actual run time class of the object that is calling the method
  - search the class for that method
  - if found, execute it
  - otherwise, go to the super class and keep looking
  - repeat until a version is found
- Is it possible the runtime system won't find a method?

Question 2

What is output by the code to the right when run?

A. `!!live`

B. `!eggegg`

C. `!egglive`

D. `!!!`

E. `eggegglive`

```
public class Animal
{
  public String bt(){ return "!"; }
}
public class Mammal extends Animal
{
  public String bt(){ return "live"; }
}
public class Platypus extends Mammal
{
  public String bt(){ return "egg";}
}
Animal a1 = new Animal();
Animal a2 = new Platypus();
Mammal m1 = new Platypus();
System.out.print( a1.bt() );
System.out.print( a2.bt() );
System.out.print( m1.bt() );
```

Question 2

What is output by the code to the right when run?

A. `!!live`

B. `!eggegg`

C. `!egglive`

D. `!!!`

E. `eggegglive`

```
public class Animal
{
  public String bt(){ return "!"; }
}
public class Mammal extends Animal
{
  public String bt(){ return "live"; }
}
public class Platypus extends Mammal
{
  public String bt(){ return "egg";}
}
Animal a1 = new Animal();
Animal a2 = new Platypus();
Mammal m1 = new Platypus();
System.out.print( a1.bt() );
System.out.print( a2.bt() );
System.out.print( m1.bt() );
```

Generic Types

An Array-Based List

- Started with a list of **`int`**s
- 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[] myCon;
```

Using the **`Object`** Class

- In Java, all classes inherit from exactly one other class, except **`Object`** which is at the top of the class hierarchy
- **`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, etc.

Difficulties with `Object`

- *Creating* generic containers using the **`Object`** data type and polymorphism is relatively straight forward
- Using these generic containers leads to some difficulties
  - Casting
  - Type checking

Question 1

- What is output by the following code?

```
GenericList list = new GenericList(); // 1
String name = "Olivia";
list.add(name); // 2
System.out.print( list.get(0).charAt(2) );// 3
```

A. `i`

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.

Question 1

- What is output by the following code?

```
GenericList list = new GenericList(); // 1
String name = "Olivia";
list.add(name); // 2
System.out.print( list.get(0).charAt(2) );// 3
```

A. `i`

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

```
ArrayList li = new ArrayList();
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 Strings
public void printFirstChar(ArrayList li) {
      String temp;
      for(int i = 0; i < li.size(); i++) {
            temp = (String)li.get(i);
            if( temp.length() > 0 )
                  System.out.println(
                        temp.charAt(0) );
      }
}
// what happens if pre condition not met?
```

Too Generic?

- Does the compiler allow this?

```
ArrayList list = new ArrayList();
list.add( "Olivia" );
list.add( new Integer(12) );
list.add( new Rectangle() );
list.add( new ArrayList() );
```

A. Yes

B. No

Too Generic?

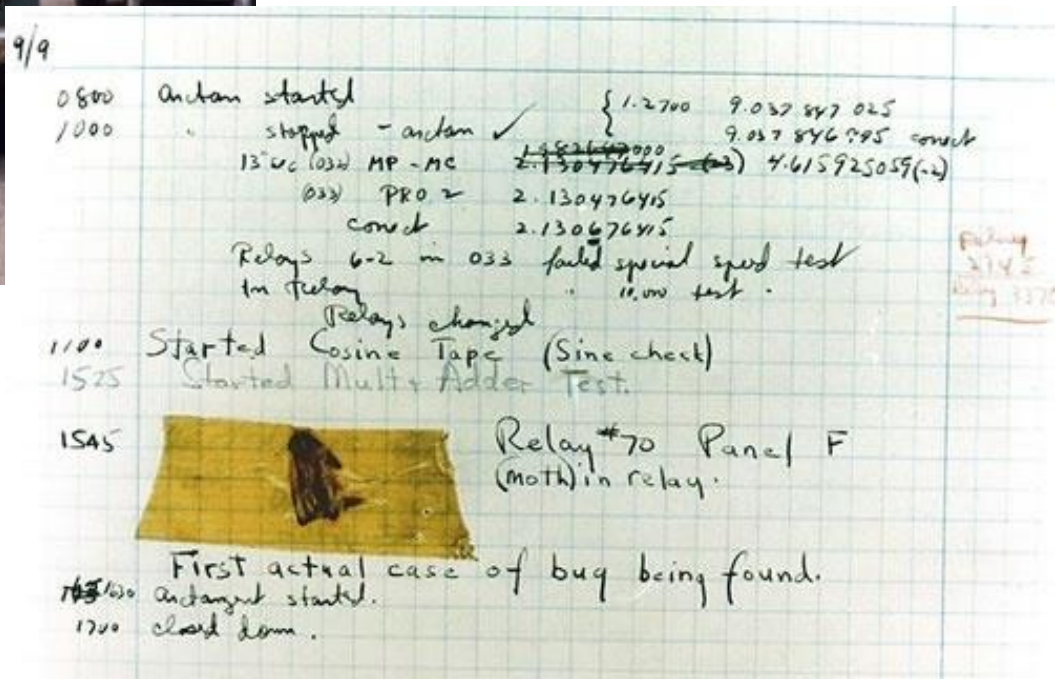
- Does the compiler allow this?

```
ArrayList list = new ArrayList();
list.add( "Olivia" );
list.add( new Integer(12) );
list.add( new Rectangle() );
list.add( new ArrayList() );
```

A. Yes

B. No

Is this a Bug or a Feature?



"Fixing" the Method

```
//pre: all elements of li are Strings
public void printFirstChar(ArrayList li) {
      String temp;
      for(int i = 0; i < li.size(); i++) {
            if( li.get(i) instanceof String ){
                  temp = (String)li.get(i);
                  if( temp.length() > 0 )
                        System.out.println(
                              temp.charAt(0) );
            }
      }
}
```

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
  - this is an abstraction. Actually, all data type info is erased at compile time

Making our Array List Generic

- Data type variables declared in class header

```
public class GenericList<T>
```

- The `<T>` is the declaration of a data type parameter for the class
  - any legal identifier:

```
Foo, AnyType, String, ClosedShape, Circle
```

- The value **T** stores will be filled in whenever a programmer creates a new

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

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

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

Why Bother?

- Polymorphism allows code reuse in another way
- Inheritance and polymorphism allow programmers to create *generic data structures*