

Generics

CIS 120 Lab 6

What is a generic class?

- ♦ A generic class is a class that can perform operations on instances of some other arbitrary class which is determined at runtime.
 - ♦ In effect, the generic class can be parameterized by another class.
- ♦ This is useful because it helps write code which is less situation dependent, leading to less repetition (and hopefully complexity) of code.
- ♦ As an example, consider a class which basically holds a pair of Strings.

```
public class Pair{
    String first;
    String second;

    public Pair(String n1, String n2){
        first=n1;
        second=n2;
    }

    public String getFirst(){
        return first;
    }

    public String getSecond(){
        return second;
    }

    public void swap(){
        String tmp=first;
        first=second;
        second=tmp;
    }
}
```

But what if we also want to be able to represent a pair of Integers?

We have to write another class.

```
public class Pair{
    Integer first;
    Integer second;

    public Pair(Integer n1, Integer n2){
        first=n1;
        second=n2;
    }

    public Integer getFirst(){
        return first;
    }

    public Integer getSecond(){
        return second;
    }

    public void swap(){
        Integer tmp=first;
        first=second;
        second=tmp;
    }
}
```

```
public class Pair{
    String first;
    String second;

    public Pair(String n1, String n2){
        first=n1;
        second=n2;
    }

    public String getFirst(){
        return first;
    }

    public String getSecond(){
        return second;
    }

    public void swap(){
        String tmp=first;
        first=second;
        second=tmp;
    }
}
```

- Does this seem silly to anyone else?
- We just wrote two classes where the only difference was “String” vs. “Integer”. This is obviously a waste of time.
 - Also, imagine if the classes were more complex [like if they had methods `setFirst()`, `setSecond()`, `sum()`, or `clear()`]
- Surely there is a more efficient way to do this.

```
public class Pair<E> {  
    E first;  
    E second;  
  
    public Pair(E n1, E n2){  
        first=n1;  
        second=n2;  
    }  
  
    public E getFirst(){  
        return first;  
    }  
  
    public E getSecond(){  
        return second;  
    }  
  
    public void swap(){  
        E tmp=first;  
        first=second;  
        second=tmp;  
    }  
}
```

```
public class Pair{  
    String first;  
    String second;  
  
    public Pair(String n1, String n2){  
        first=n1;  
        second=n2;  
    }  
  
    public String getFirst(){  
        return first;  
    }  
  
    public String getSecond(){  
        return second;  
    }  
  
    public void swap(){  
        String tmp=first;  
        first=second;  
        second=tmp;  
    }  
}
```

We've just defined a Pair that can hold any type of object that we can define at runtime.

Here's how we could use the improved Pair class:

```
> Pair<String> p1 = new Pair<String>("happy", "birthday");  
>p1.getFirst()  
"happy"
```

```
>Pair<Integer> p2 = new Pair<Integer>(4,2);  
>p2.getFirst()  
4
```

Note, we can use the same Pair class for two different types of objects.

Now, let's try to create a file called Triplet.java and write in the following functionality:

- It should be parameterized by a single class
- It should have a constructor which takes 3 instances of the parameterizing class and stores them internally
- The following methods should be implemented in a logical manner
 - `getFirst()`, `getSecond()`, and `getThird()`
 - `shiftLeft()` [shift elements to left by 1 slot: 3->2, 2->1, 1->3]
 - `toString()` [should print out the elements in order]
- Next, add a constructor which takes a reference to another Triplet and creates the new Triplet based on the elements of the old one

For the next few minutes, think about (and maybe try to write) a `Triplet.java` that adheres to the specifications on the previous page.

Download the `TripletTester.java` file from the course website to test your `Triplet`.

If you finish early, try to write a `shift(int x)` method where `x` specifies how many spaces to shift your elements.

My output:

First, making a Triplet of Strings

Before shiftLeft():

First ele: 'hello', second ele: 'there', third ele: 'world'

After shiftLeft():

First ele: 'there', second ele: 'world', third ele: 'hello'

Now with a triplet of ints

Before shiftLeft():

First ele: '3', second ele: '1', third ele: '4'

After shiftLeft():

First ele: '1', second ele: '4', third ele: '3'

And now let's make a new one based off this:

First ele: '1', second ele: '4', third ele: '3'

Now with a triplet of Exceptions

Before shiftLeft():

First ele: 'java.lang.IllegalArgumentException: We noticed an arg breaking the law', second ele: 'java.lang.NullPointerException: Couldn't find my head', third ele: 'java.lang.ArithmeticException: 2+2!=4'

After shiftLeft():

First ele: 'java.lang.NullPointerException: Couldn't find my head', second ele: 'java.lang.ArithmeticException: 2+2!=4', third ele: 'java.lang.IllegalArgumentException: We noticed an arg breaking the law'

Notice that in the last test, the Tester inserted some classes we don't even know about into our Triplet. Even more importantly, notice that our Triplet behaved in the exact same way as usual.

This is one very strong advantage of generics. They can easily deal with classes that will be unknown until runtime.

Let's take a look at one possible way of implementing this Triplet class.

```
public class Triplet<E>{

    private E first;
    private E second;
    private E third;

    public Triplet(E n1, E n2, E n3){
        first=n1;
        second=n2;
        third=n3;
    }

    public Triplet(Triplet<E> copy){ //copy must be parameterized by the same class E
        first=copy.getFirst();
        second=copy.getSecond(); //so we can easily copy its information
        third=copy.getThird();
    }

    public E getFirst(){
        return first;
    }

    public E getSecond(){
        return second;
    }

    public E getThird(){
        return third;
    }
}
//more on next page
```

//continued from previous page

```
public void shiftLeft(){  
    E tmp=first;  
    first=second;  
    second=third;  
    third=tmp;  
}
```

```
public void shift(int n){  
    n%=3;  
    n-=3;  
    for(int i=0; i>n;i--)  
        shiftLeft();  
}
```

```
public String toString(){  
    return "First ele: "+first+", second ele: "+second+", third ele: "+third+"";  
}
```

```
}
```

Let's try to write another example that uses generics.

Consider a dictionary class that is basically a list of entries, where an entry is some lookup item (often a String) connected to a piece of information (some other object).

The following code should run:

```
> Dictionary<String, Integer> ages = new Dictionary<String, Integer>();  
> ages.add("me", 19);  
> ages.add("mom", 40);  
> ages.lookup("me");  
19
```

```
> Dictionary<String, String> names = new Dictionary<String, String>();  
> names.add("mom", "Hope");  
> names.add("dad", "Rick");  
> names.lookup("mom");  
"Hope"
```

Write a class Dictionary.java that satisfies the example given on the previous slide.

Specifically, create a class Dictionary which can be parameterized by two separate classes, u and v , such that a mapping between the two is stored and that for any pair (u_i, v_i) added, $\text{lookup}(u_i)$ returns v_i .

If you're feeling particularly ambitious, try to add an additional method $\text{remove}(u)$ that removes an entry.

As a hint, consider that the Dictionary is basically a list of entries, or pairs of Objects. As such, it may be useful to write a helper class that represents an entry.

A sample solution will be posted online after lab.