Ice Breaker

What is your study ~vibe~?

Exam 2 Review

CSE 017 | Prof Ziltz | Fall 22

Comparable

- Defined in `java.lang` (example below):
- `interface` used to compare objects of any type/class
- Includes only one abstract method: `compareTo()`
  - Override in classes that `implement Comparable` to mimic the following:
    - returns 0 if the two arguments are equal
    - returns > 0 if the first argument comes after the second argument
    - returns < 0 if the first argument comes before the second argument

```
public interface Comparable<E> {
   int compareTo(E obj);
}
```

Creating Comparable Shapes

- Java doesn't allow for multiple inheritance
- Alt: can implement a list of interfaces (sep by commas)

```
public interface Comparable<E> {
   int compareTo(E obj);
}
```

```
public interface Cloneable {

}
```

```
public class Circle extends Shape
          implements Comparable<Circle>, Cloneable {
   …
   public int compareTo(Circle cc){
      if (radius == cc.radius) return 0;
      else if (radius > cc.radius) return 1;
      else return -1;
   }
   public Object clone(){
     return new Circle(radius));
   }
}
```

Using Comparator to sort Shapes

<<Interface>>
java.util.Comparator;

```
int compare(T obj1, T obj2);
boolean equals(T obj);
```

java.util.Arrays;

```
<E> void sort(E[] list,Comparator<? Super E> c)
```

```
public class comparatorByColor implements Comparator<Shape> {
    int compare(Shape s1, Shape s2){
      return s1.getColor().compareTo(s2.getColor());
    }
}
```

```
public class comparatorByArea implements Comparator<Shape> {
      int compare(Shape s1, Shape s2){
      Double area1 = s1.getArea();
      Double area2 = s2.getArea();
      return area1.compareTo(area2);
    }
}
```

Using Comparator to sort Shapes

```
public class comparatorByColor implements Comparator<Shape> {
    int compare(Shape s1, Shape s2){
      return s1.getColor().compareTo(s2.getColor());
    }
}
```

```
public class comparatorByArea implements Comparator<Shape> {
       int compare(Shape s1, Shape s2){
      Double area1 = s1.getArea();
      Double area2 = s2.getArea();
      return area1.compareTo(area2);
    }
}
```

```
public static void main(String[] args) {
          Shape[] s = {new Circle(),
                 new Circle("Red", 5.0),
                 new Circle("Blue", 2.5),
                 new Rectangle(),
                 new Rectangle("Green", 10.5, 5.5),
                 new Rectangle("Yellow", 4.0, 2.5)};
          printArray(s);
          System.out.println("\n");
          java.util.Arrays.sort(s, new ComparatorByArea());
          printArray(s);
          System.out.println("\n");
          java.util.Arrays.sort(s, new ComparatorByColor());
          printArray(s);
     }
```

Using Generic Classes/Interfaces

- **Generic Class - Class of type `<E>`**
- **`E` is the type parameter or generic type**
- **`E` can be replaced by any reference type String, Integer, or Student**
- **Primitive types are not allowed as generic type parameters (int, double, char, ...)**
  - This is why we care about wrapper classes ^
- **Can use any name for the generic type (between <>) but the convention is <E> or <T>**

Creating Generic Classes

Stack<E>
-elements: ArrayList<E>
+Stack() +push(E item): void +pop(): E +peek(): E +isEmpty(): boolean +size(): int +toString(): String

Creating a Generic Class: Stack <E>

```
import java.util.ArrayList;
public class Stack<E> {
	private ArrayList<E> elements;
	public Stack() {
		elements = new ArrayList<>();
	}
	public int size() {
		return elements.size();
	}
	public boolean isEmpty() {
		return elements.isEmpty();
	}
	public void push(E item) {
		elements.add(item);
	}
	public E peek() {
		return elements.get(size()-1);
	}
	public E pop() {
		E item = elements.get(size()-1);
		elements.remove(size()-1);
		return item;
	}
	public String toString() {
		return "Stack: " + elements.toString();
	}
}
```

```
public class TestStack {
   public static void main(String[] args) {

       Stack<String> cityStack = new Stack<>();
       cityStack.push("New York");
       cityStack.push("London");
       cityStack.push("Paris");
       cityStack.push("Tokyo");
       System.out.println("Stack of Cities");
       System.out.println(cityStack.toString());
       System.out.println("Top element: " + cityStack.peek());
   }
}
```

Creating a Generic Class: Stack <E>

```
import java.util.ArrayList;
public class Stack<E> {
	private ArrayList<E> elements;
	public Stack() {
		elements = new ArrayList<>();
	}
	public int size() {
		return elements.size();
	}
	public boolean isEmpty() {
		return elements.isEmpty();
	}
	public void push(E item) {
		elements.add(item);
	}
	public E peek() {
		return elements.get(size()-1);
	}
	public E pop() {
		E item = elements.get(size()-1);
		elements.remove(size()-1);
		return item;
	}
	public String toString() {
		return "Stack: " + elements.toString();
	}
}
```

```
public class TestStack2{
   public static void main(String[] args) {
        Stack<Integer> numberStack = new Stack<>();
        numberStack.push(11);
        numberStack.push(22);
        numberStack.push(33);
        numberStack.push(44);
        numberStack.push(55);

        System.out.println("Stack of numbers");
        System.out.println(numberStack.toString());
        System.out.println("Top element: " +numberStack.peek());
    }
}
```

Implementing Generics

- **After compile time, `E` is removed and replaced with the raw type (`Object`)**
  - Erasure of the generic type
- **Old way of implementing generics: use type `Object` instead of `E`**
  - Using array with elements of type `Object` would also work
- **Using Generics improves software reliability and readability**
- **Errors are detected at compile time**
  - This is why we use `E` (a concrete type) rather than `Object` (ex follows)

ObjectStack versus Stack<String>

```
public class ObjectStack {
  private Object[] elements;
  int size;
  public ObjectStack() {
     elements = new Object[10]; size = 0;}
  public void push(Object item) { elements[size++] = item; }
  public Object peek() { return elements[size-1]; }
  public int size() { return size; }
  public Object pop() { return elements[--size]; }
  public boolean isEmpty() { return (size == 0); }
  public String toString(){
     String s = "Stack: [";
     int i=0;
     for( ; i<size-1; i++){
        s+= elements[i].toString() + "";
     }
     s+= elements[i].toString() + "]";
     return s;}}
```

```
public class TestObjectStack {
   public static void main(String[] args) {
       ObjectStack cityStack = new ObjectStack();
       cityStack.push("New York");
       cityStack.push("London");
       cityStack.push("Paris");
       cityStack.push("Tokyo");
       cityStack.push(22); // ok
       System.out.println("Stack of Cities\n" + cityStack.toString());
       System.out.println("Top element: " + cityStack.peek());
   }
}
```

```
public class TestStack {
   public static void main(String[] args) {

       Stack<String> cityStack = new Stack<>();
       cityStack.push("New York");
       cityStack.push("London");
       cityStack.push("Paris");
       cityStack.push("Tokyo");
       cityStack.push(22);
       System.out.println("Stack of Cities");
       System.out.println(cityStack.toString());
       System.out.println("Top element: " +
             cityStack.peek());
   }
}
```

The method push(String) in the type Stack<String> is not applicable for the arguments (int)

Testing Multiple Generic Types

```
public class Pair<E1, E2> {
   private E1 first;
   private E2 second;
   public Pair(E1 first, E2 second) {
       this.first = first;
       this.second = second;
   }
   public void setFirst(E1 first) {
       this.first = first;
   }
   public void setSecond(E2 second) {
       this.second = second;
   }
   public E1 getFirst() {
       return first;
   }

   public E2 getSecond() {
       return second;
   }
   public String toString() {
       return "(" + first.toString() + ", " +
              second.toString() + ")";
   }
   public boolean equals(Object obj) {
       Pair<E1, E2> p = (Pair<E1, E2>) obj;
       boolean eq1 = p.getFirst().equals(first);
       boolean eq2 = p.getSecond().equals(second);
       return eq1 && eq2;
   }
}
```

```
import java.util.ArrayList;
public class TestPair {
   public static void main(String[] args) {
       ArrayList<Pair<Integer, String>> list = new ArrayList<>();
       Pair<Integer, String> p;
       p = new Pair<Integer, String>(12345, "Lisa Bello");
       list.add(p);
       p = new Pair<Integer, String>(54321, "Karl Johnson");
       list.add(p);
       p = new Pair<Integer, String>(12543, "Jack Green");
       list.add(p);
       p = new Pair<Integer, String>(53241, "Emma Carlson");
       list.add(p);
       System.out.println(list.toString());

   }
}
```

Generic Methods

- **A method can be generic: parameters or return value are of type generic**
  - Does not have to be in a generic class
- **Printing arrays of different types `printArray()`**
- **Searching arrays of different types**
- **Sorting arrays of different types `java.util.Arrays.sort()`**

Generic `printArray()`

```
public class GenericPrint{
   public static void main(String[] args) {
       Integer[] numbers = {11, 22, 33, 44, 55};
       String[] names = {"Kallie", "Brandon", "Amelia", "Doug"};
       printArray(numbers);
       printArray(names);
   }
   public static <E> void printArray(E[] list) {
       System.out.print("[ ");
       for (int i=0; i<list.length; i++)
           System.out.print(list[i] + " ");
       System.out.println("]");
   }

}
```

Generic `sortArray()`

- Sorting arrays of different types `java.util.Arrays.sort()`
- `sort()` needs to compare the elements (order them)
- Elements of the array need to be compared – must be comparable
  - Restrict the generic method to objects that can call compareTo()

```
public static <E extends Comparable<E>> void sort(E[] list) {
// Selection Sort
       int currentMinIndex;
         E currentMin;
         for (int i=0; i<list.length-1; i++) {
             currentMinIndex = i;
             currentMin = list[i];
             for(int j=i+1; j<list.length; j++) {
                 if(currentMin.compareTo(list[j]) > 0) {
                     currentMin = list[j];
                     currentMinIndex = j;
                 }
             }
             if (currentMinIndex != i) {
                 list[currentMinIndex] = list[i];
                 list[i] = currentMin;}
         }
  }
```

Generic sortArray()

```
public static <E extends Comparable<E>> void sort(E[] list) {
// Selection Sort
       int currentMinIndex;
         E currentMin;
         for (int i=0; i<list.length-1; i++) {
             currentMinIndex = i;
             currentMin = list[i];
             for(int j=i+1; j<list.length; j++) {
                 if(currentMin.compareTo(list[j]) > 0) {
                     currentMin = list[j];
                     currentMinIndex = j;
                 }
             }
             if (currentMinIndex != i) {
                 list[currentMinIndex] = list[i];
                 list[i] = currentMin;}
         }
  }
```

```
public class GenericSort{
   public static void main(String[] args) {
       Integer[] numbers = {11, 22, 33, 44, 55};
       String[] names = {"Kallie", "Brandon", "Amelia", "Doug"};
       sortArray(numbers);
       sortArray(names);
       printArray(numbers);
       printArray(names);
   }
   public static <E> void printArray(E[] list) {
       System.out.print("[ ");
       for (int i=0; i<list.length; i++)
           System.out.print(list[i] + " ");
       System.out.println("]");
   }
```

Restrictions on Generics

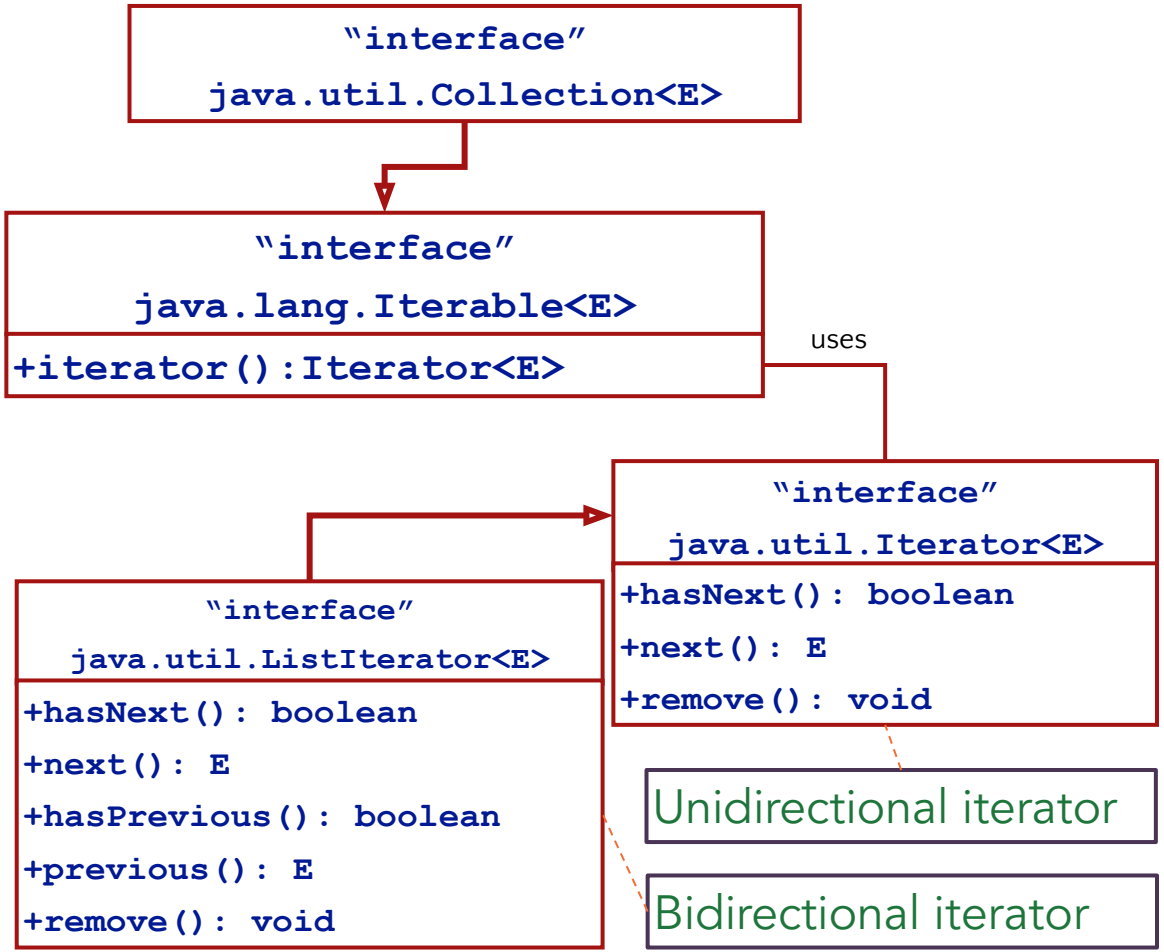
1. **Cannot create instances using the generic type <E>**
   a. The following is incorrect: `E item = new E();`
2. **Cannot create an array of type E**
   a. The following is incorrect: `E[] list = new E[20];`
3. **Generic type is not allowed in a static context**
   a. All instances of a generic class share same runtime class
   b. The following are incorrect:

```
public static E item;
public static void m(E object)
```

4. **Exceptions cannot be Generic**
   a. The following are incorrect:

```
public class MyException<T> extends Exception{ }
public static void main(String[] args){
  try{
     Cannot check the thrown exception
  }
  catch(MyException<T> ex){
  }
}
```

Java Collection Framework: Iterators



Iterator Example

```
import java.util.ArrayList;
import java.util.Iterator;
public class Test1{
   public static void main(String[] args) {
       ArrayList<String> al = new ArrayList<>();
       al.add("New York"); al.add("Tokyo");
       al.add("Paris"); al.add("Rome");
       al.add("Brasilia");
       Iterator<String> iter = al.iterator();
       System.out.print("[ ");
       while(iter.hasNext()){
           System.out.print(iter.next().toUpperCase() + " ");
       }
       System.out.print("]");
   }
}
```

```
[ NEW YORK TOKYO PARIS ROME BRASILIA ]
```

S21 ContainsPoint()

```
Returns true if p is found in the list of points.
This method should be recursive, use linear search, and use an iterator to visit the elements of
the list
```

S21 ContainsPoint()

```
public boolean containsPoint(Pair<Integer, Integer> point) {
            Iterator<Pair<Integer, Integer>> iter = listOfPoints.iterator();
            return containsPoint(point, iter);
      }
```

```
public boolean containsPoint(Pair<Integer, Integer> point, Iterator<Pair<Integer, Integer>> iter)
{
            if (iter.hasNext()) {
                  if(iter.next().equals(point))
                        return true;
                  else
                        return containsPoint(point, iter);
            }
            return false;

      }
```