Generics

CSCI 201
Principles of Software Development

Jeffrey Miller, Ph.D.
jeffrey.miller@usc.edu

USC Viterbi
School of Engineering

Outline

- Generics
  - Generic Classes
  - Generic Methods
- Program

Generics Overview

- Generics were introduced in Java version 5.0 in 2004
- Generics allow you to parameterize types
  - This is similar to templates in C++
- A class or method can be defined with generic types that the compiler can replace with concrete types
- Generics allow type errors to be detected at compile time rather than runtime
  - A generic class or method allows you to specify allowable types of objects that the class or method can use, and if you attempt to use an incompatible type, a **compiler** error will occur
  - Before generics, methods that took multiple types had to take a parent type (usually `Object`), but then any variable that inherits from that class could be passed in, not just a subset of them, potentially causing a **runtime** error

```
1  public interface Comparable {
2    public int compareTo(Object o);
3    ...
4  }
```

```
1  public interface Comparable<T> {
2    public int compareTo(T o);
3    ...
4  }
```

Generic Instantiation

- To create a class or interface that takes a generic, include an identifier (by convention, a single capital letter) in angle brackets immediately following the name of the class or interface
  - Use that identifier in any variable declaration or return type in the class
  - When you create an instance of that class, you will need to parameterize the class with a concrete class to use in place of that generic
    - Note: You must use a concrete class and not a primitive data type or abstract class

```
1   public class Test<T> {
2
3     public void print(T t) {
4       System.out.println(t);
5     }
6
7     public static void main(String args[]) {
8       Test<String> t = new Test<String>();
9       t.print("Hello CSCI201");
10    }
11  }
```

Using Generics

- Does this code compile?

```
1   public class Test<T> {
2
3     public void print(T t) {
4       System.out.println(t);
5     }
6
7     public static void main(String args[]) {
8       Test<String> t = new Test<String>();
9       t.print(33);
10    }
11  }
```

Generics with `ArrayList`

- The generic passed into an `ArrayList` must be an object, not a primitive type
  - Casting is allowed though

```
1   import java.util.ArrayList;
2
3   public class Test {
4
5     public static void main(String args[]) {
6       ArrayList<Integer> arr = new ArrayList<Integer>();
7       arr.add(3);
8       arr.add(new Integer(4));
9       arr.add("5");
10      int num = arr.get(0);
11      System.out.println("num = " + num);
12    }
13  }
```

This is called autoboxing, which is a type of casting that takes a primitive type and converts it to its wrapper object

Generic Stack

```
1  import java.util.ArrayList;
2  public class Test<E> {
3    private ArrayList<E> list =
       new ArrayList<E>();
4
5    public int getSize() {
6     return list.size();
7    }
8    public E peek() {
9     return list.get(getSize()-1);
10   }
11   public void push(E o) {
12    list.add(o);
13   }
14   public E pop() {
15    E o = list.get(getSize()-1);
16    list.remove(getSize()-1);
17    return o;
18   }
```

```
19  public boolean isEmpty() {
20   return list.isEmpty();
21  }
22  public static void main(String [] args) {
23   Test<String> stack1 = new Test<String>();
24   stack1.push("CSCI103");
25   stack1.push("CSCI104");
26   stack1.push("CSCI201");
27   Test<Integer> stack2 = new Test<Integer>();
28   stack2.push(103);
29   stack2.push(104);
30   stack2.push(201);
31  }
32 }
```

Multiple Generics

```
1   import java.util.ArrayList;
2   public class GenericStack<E, F> {
3     private ArrayList<E> list =
        new ArrayList<E>();
4
5     public int getSize() {
6       return list.size();
7     }
8     public E peek() {
9       return list.get(getSize()-1);
10    }
11    public void push(E o) {
12      list.add(o);
13    }
14    public E pop() {
15      E o = list.get(getSize()-1);
16      list.remove(getSize()-1);
17      return o;
18    }
```

```
19    public boolean isEmpty() {
20      return list.isEmpty();
21    }
22    public void print(F f) {
23      System.out.println(f);
24    }
25    public static void main(String [] args) {
26      GenericStack<String, Double> stack1 =
          new GenericStack<String, Double>();
27      stack1.push("CSCI103");
28      stack1.push("CSCI104");
29      stack1.push("CSCI201");
30      stack1.print(3.5);
31      GenericStack<Integer, String> stack2 =
          new GenericStack<Integer, String>();
32      stack2.push(103);
33      stack2.push(104);
34      stack2.push(201);
35      stack2.print("Hello CSCI201");
36    }
37  }
```

Generic Methods

- If you want to use a generic in a `static` method within a class, the following syntax is used
  - This is because there is no instantiation needed to call a `static` method, so we needed to be able to parameterize the method call

```
1   public class Test {
2     public static<T> void print(T [] list) {
3       for (int i=0; i < list.length; i++) {
4         System.out.print(list[i] + " ");
5       }
6     }
7     public static void main(String[] args ) {
8        Integer[] integers = {1, 2, 3, 4, 5};
9        String[] strings = {"London", "Paris", "New York"};
10
11       Test.<Integer>print(integers);
12       Test.<String>print(strings);
13    }
14  }
```

Bounded Generics

- You can specify that you want a generic type to be a subtype of another type

```
1   import java.io.BufferedReader;
2   import java.io.FileReader;
3   import java.io.IOException;
4   import java.io.Reader;
5
6   public class Test {
7     public static<T extends Reader> void openFile(T t) throws IOException {
8       BufferedReader br = new BufferedReader(t);
9       String line = br.readLine();
10    }
11    public static void main(String[] args ) {
12      try {
13        Test.<FileReader>openFile(new FileReader(args[0]));
14      } catch (IOException ioe) {
15        System.out.println("exception: " + ioe.getMessage());
16      }
17    }
18  }
```

Missing Generics

- If a class or method is parameterized and you leave off the generic instantiation, the class or method will be parameterized with the `Object` class
  - This is why you cannot use primitive types in generic instantiations
  - The following two lines are equivalent
  - This keeps backwards compatibility with previous versions of Java before serialization existed

```
1  Vector v = new Vector();
2  Vector<Object> v = new Vector<Object>();
```

Subclass Generics

- Does the following code compile?

```
1   public class WildCardDemo {
2     public static void main(String[] args ) {
3       GenericStack<Integer> intStack = new GenericStack<Integer>();
4       intStack.push(1); // autoboxing
5       intStack.push(2); // autoboxing
6       intStack.push(-2); // autoboxing
7       System.out.print("The max number is " + max(intStack));
8     }
9     public static double max(GenericStack<Number> stack) {
10      double max = stack.pop().doubleValue(); // Initialize max
11      while (!stack.isEmpty()) {
12        double value = stack.pop().doubleValue();
13        if (value > max) {
14          max = value;
15        }
16      }
17      return max;
18    }
19  }
```

Wildcard Generics

- There are three types of wildcard generics
  - Unbounded wildcard – written either as `?` or `? extends Object`
  - Bounded wildcard – written as `? extends T` represents T or an unknown subtype of T
  - Lower-bound wildcard – written as `? super T` denotes T or an unknown supertype of T
- Wildcard generics can be used on parameterized classes and parameterized methods

Subclass Generics

- This code now compiles

```
1   public class WildCardDemo {
2     public static void main(String[] args ) {
3       GenericStack<Integer> intStack = new GenericStack<Integer>();
4       intStack.push(1); // 1 is autoboxed into new Integer(1)
5       intStack.push(2);
6       intStack.push(-2);
7       System.out.print("The max number is " + max(intStack));
8     }
9     public static double max(GenericStack<? extends Number> stack) {
10      double max = stack.pop().doubleValue(); // Initialize max
11      while (!stack.isEmpty()) {
12        double value = stack.pop().doubleValue();
13        if (value > max) {
14          max = value;
15        }
16      }
17      return max;
18    }
19  }
```

Superclass Generics

- Does the following code compile?

```
1   public class WildCardDemo3 {
2     public static void main(String[] args) {
3       GenericStack<String> stack1 = new GenericStack<String>();
4       GenericStack<Object> stack2 = new GenericStack<Object>();
5       stack2.push("Java");
6       stack2.push(2);
7       stack1.push("Sun");
8       add(stack1, stack2);
9     }
10
11    public static<T> void add(GenericStack<T> s1, GenericStack<T> s2) {
12      while (!s1.isEmpty()) {
13        s2.push(s1.pop());
14      }
15    }
16  }
```

Superclass Generics (cont.)

- The following code now compiles

```
1   public class WildCardDemo3 {
2     public static void main(String[] args) {
3       GenericStack<String> stack1 = new GenericStack<String>();
4       GenericStack<Object> stack2 = new GenericStack<Object>();
5       stack2.push("Java");
6       stack2.push(2);
7       stack1.push("Sun");
8       add(stack1, stack2);
9     }
10
11    public static<T> void add(GenericStack<T> s1, GenericStack<? super T> s2) {
12      while (!s1.isEmpty()) {
13        s2.push(s1.pop());
14      }
15    }
16  }
```

Outline

- Generics
  - Generic Classes
  - Generic Methods
- Program

Program

- Create a sorting program that allows any type of number to be taken as a parameter and it will still sort the array. The user should be prompted to enter the type of variables he wants.

```
(i) Integer
(f) Float
(d) Double
What type of array would you like to sort? f
Original array: 3.4, 1.2, 5.2, 6.3, 2.9
Sorted array: 1.2, 2.9, 3.4, 5.2, 6.3
```