Advanced Programming
CSE 201
Instructor: Sambuddho

(Semester: Monsoon 2024)
Week 5 – Generic Classes

Generic Container to Hold Different Types ?

- By using any of the concepts taught till now in this course, how can you store different types of objects in a same datastructure
  - E.g., String, Integer, Float, etc. ?

Approach 1

```
public class MyGenericList { private
    ArrayList myList; public
    MyGenericList() {
       myList = new ArrayList();
    }
    public void add(Object o)
       { myList.add(o);
    }
    public Object get(int i) { return
       myList.get(i);
    }



    public static void main(String[] args)
      { MyGenericList generic = new
MyGenericList();
      generic.add("hello"); generic.add(10);
      generic.add(10.23f);
      ......
      String str = (String) generic.get(0); // OK
      String str = (String) generic.get(1); //
NOT OK


    }
```

- Using inheritance we know Object class can hold any type of objects
  - We can create ArrayList of objects
- Problems we face:
  - Mandatory type casting while getting the object from list
  - No error checking while adding objects as we are allowed to add any type of objects
    - Wrong type casting can land with runtime errors

Generic Programming

- Code that can be reused. Need not be rewritten for individual types.
- Same class and methods can be used for multiple types (non primitive).
- Avoid generic casting errors.

```
public class ArrayList // before generic classes
{
   private Object[] elementData;
   . . .
   public Object get(int i) { . . . }
   public void add(Object o) { . . . }
}
```

```
ArrayList files = new ArrayList();
files.add(new File(". . ."));
String filename = (String) files.get(0);
```

Solution: Generic Programming

- Our generic cup can hold different types of liquid
- In the notation Cup<T>:
  - T = Coffee
  - T = Tea
  - T = Milk T
  - T = Soup
  - … …

Cup == Generic Container

Generic Programming

```
var files = new ArrayList<String>();
```

Or

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

```
public class Pair<T, U> { . . . }
```

```
public class Pair<T>
{
   private T first;
   private T second;

   public Pair() { first = null; second = null; }
   public Pair(T first, T second) { this.first = first; this.second = second; }

   public T getFirst() { return first; }
   public T getSecond() { return second; }

   public void setFirst(T newValue) { first = newValue; }
   public void setSecond(T newValue) { second = newValue; }
}
```

Generic Methods

```
class ArrayAlg
{
   public static <T> T getMiddle(T... a)
   {
      return a[a.length / 2];
   }
}
```

```
String middle = ArrayAlg.<String>getMiddle("John", "Q.", "Public");
```

```
String middle = ArrayAlg.getMiddle("John", "Q.", "Public");
```

Solution to our Problem

```
public class MyGenericList <T>
    { private ArrayList <T> myList;
    public MyGenericList() {
       myList = new ArrayList
<T>();
    }
    public void add(T o)
       { myList.add(o);
    }
    public T get(int i) { return
       myList.get(i);
    }
}
```

- Using generic programming we don't have to implement different classes for different object types.
  - Programmer friendly code!
- We just have to create different instances of MyGenericLists.

```
public class Main {
         public static void main(Sting args[]) {
           MyGenericList<String> strList = new
MyGenericList<String>();
        MyGenericList<Integer> intList = new
MyGenericList<Integer>();


         strList.add("hello");
         intList.add(1);
         ...
    }
```

Generic Class with Two Fields (1/3)

```
public class Pair <T1, T2> { private T1 key;
    private T2 value;
    public Pair(T1 _k, T2 _v) { key = _k; value
       = _v;
    }
    public T1 getKey() {       return key; }
    public T2 getValue() { return value; }
}
```

```
public class Main {
    public static void main(Sting args[]) {
        MyGenericList<Pair> db =
             new MyGenericList<Pair>();
        db.add(new Pair<String, Integer>(“John”, 2343));
        db.add(new Pair<String, Integer>(“Susane”, 8908));
        ...
    }
}
```

- Why this code isn’t correct?
  - MyGenericList class instantiated without specifying the type of its two fields

Generic Class with Two Fields (2/3)

```
public class Pair <T1, T2> { private T1 key;
    private T2 value;
    public Pair(T1 _k, T2 _v) { key = _k; value
       = _v;
    }
    public T1 getKey() {       return key; }
    public T2 getValue() { return value; }
}
```

```
public class Main {
    public static void main(Sting args[]) {
        MyGenericList<Pair<String, Integer>> db = new
             MyGenericList<Pair>();
        db.add(new Pair<String, Integer>("John", 2343));
        db.add(new Pair<String, Integer>("Susane", 8908));
        ...
    }
}
```

- Why this code isn't correct
  - During instantiation we have to declare the type of fields in MyGenericList class on both RHS and LHS of statement

Generic Class with Two Fields (3/3)

```
public class Pair <T1, T2> { private T1 key;
    private T2 value;
    public Pair(T1 _k, T2 _v) { key = _k; value
       = _v;
    }
    public T1 getKey() {       return key; }
    public T2 getValue() { return value; }
}
```

- This is the correct implementation and usage of a generic class with multiple fields

```
public class Main {
    public static void main(Sting args[]) {
        MyGenericList<Pair<String, Integer>> db =
             new MyGenericList<Pair<String, Integer>>();
        db.add(new Pair<String, Integer>("John", 2343));
        db.add(new Pair<String, Integer>("Susane", 8908));
        ...
    }
}
```

Why Generic Array Creation not Allowed ?

```
// Legal statement (arrays are covariant) Object array[]
= new Integer[10];
// Compilation error below (generics are invariant)

List<Object> myList = new
ArrayList<Integer>();
// Below line incorrect but let's assume its correct
List<Integer> intList[] = new
ArrayList<Integer>[5]; List<String> stringList =
new ArrayList<String>();

stringList.add("John");

Object objArray[] = intList; objArray[0] = stringList;
```

```
// This will generate ClassCastException
int my_int_number = objArray[0].get(0);
```

- Arrays are covariant
  - Subclass array type can be assigned to superclass array reference
- Generics are invariant
  - Subclass type generic type cannot be assigned to superclass generic reference.

- If generic array creation was allowed then compile time strict type checking cannot be enforced.
  - Runtime ClassCastException will

Bounds for Type Variables

```
class ArrayAlg
{
   public static <T> T min(T[] a) // almost correct
   {
      if (a == null || a.length == 0) return null;
      T smallest = a[0];
      for (int i = 1; i < a.length; i++)
         if (smallest.compareTo(a[i]) > 0) smallest = a[i];
      return smallest;
   }
}
```

Issues?
Does an object of arbitrary type have a method compareTo()?

```
public static <T extends Comparable> T min(T[] a) . . .
```

```
T extends Comparable & Serializable
```

Adding multiple bounds

Type Erasures – Basis of Generic Programming

Rule: *Erase* and replace generic type with a *raw* type (for bounded types) and *Object* for unbounded.

```
public class Pair<T>
{
   private T first;
   private T second;

   public Pair() { first = null; second = null; }
   public Pair(T first, T second) { this.first = first; this.second = second; }

   public T getFirst() { return first; }
   public T getSecond() { return second; }

   public void setFirst(T newValue) { first = newValue; }
   public void setSecond(T newValue) { second = newValue; }
}
```

```
public class Pair
{
   private Object first;
   private Object second;

   public Pair(Object first, Object second)
   {
      this.first = first;
      this.second = second;
   }

   public Object getFirst() { return first; }
   public Object getSecond() { return second; }

   public void setFirst(Object newValue) { first = newValue; }
   public void setSecond(Object newValue) { second = newValue; }
}
```

Type Erasures – Basis of Generic Programming

Rule: *Erase* and replace generic type with a *raw* type (for bounded types) and *Object* for unbounded.

```
public class Interval<T extends Comparable & Serializable> implements Serializable
{
   private T lower;
   private T upper;
   . . .
   public Interval(T first, T second)
   {
      if (first.compareTo(second) <= 0) { lower = first; upper = second; }
      else { lower = second; upper = first; }
   }
}
```

```
public class Interval implements Serializable
{
   private Comparable lower;
   private Comparable upper;
   . . .
   public Interval(Comparable first, Comparable second) { . . . }
}
```

Type Erasures – Implicit Casting

Step 1: Call to raw method Pair.getFirst();

Step 2: Cast returned object to type Object.

```
Pair<Employee> epairs = new Pair<>;
Employee epair = epairs.getFirst();
```

Type Erasures – Translating Generic Methods

```
public static <T extends Comparable> T min(T[] a)
```

```
public static Comparable min(Comparable[] a)
```

```
class DateInterval extends Pair<LocalDate>
{
   public void setSecond(LocalDate second)
   {
      if (second.compareTo(getFirst()) >= 0)
         super.setSecond(second);
   }
   . . .
}
```

```
class DateInterval extends Pair // after erasure
{
   public void setSecond(LocalDate second) { . . . }
   . . .
}
```

But Pair also has setSecond(Object second);!!

Erasure interferes with polymorphism!!

Type Erasures – Translating Generic Methods – Bridge Methods

The compiler generates a *bridge method* in the DateInterval class.

```
public void setSecond(Object second) { setSecond((LocalDate) second); }
```