CSE201: Monsoon 2020, Section-A
Advanced Programming

Lecture 10: Generic Programming

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Last Lecture

- Class Object
  - Correct implementation of `equals` method
  - Comparing objects
    - Comparable interface
    - Comparator interface
  - Copying objects

```java
public class Rectangle implements Comparable<Rectangle> {
    private int sideA, sideB, area;
    public Rectangle (int _a, int _b) { ... }
    @Override
    public int compareTo(Rectangle o) {
      if(area == o.area) return 0;
      else if(area < o.area) return -1;
      else return 1;
    }
}
```

```java
public class BankAccount implements Cloneable {
   private String name;
   private List<String> transactions;
   ...
   public BankAccount clone() {
      try {               // deep copy
         BankAccount copy = (BankAccount) super.clone();
         copy.transactions = new ArrayList<String>(transactions);
         return copy;
      } catch (CloneNotSupportedException e) {
         // this will never happen
         return null;
      }
   }
}
```

```java
public class RectangleAreaComparator
                              implements Comparator<Rectangle> {
    @Override
    public int compare(Rectangle r1, Rectangle r2) {
        return r1.getArea() - r2.getArea();
    }
}
```

```java
public class Point {
   private int x, y;
   public Point(int _x, int _y) { ... }
   @Override
    public boolean equals(Object o1) {
       if(o1 != null && getClass() == o1.getClass()) {
         Point o = (Point) o1; //type casting
         return (x==o.x && y==o.y);
       }
       else {
         return false;
       }
    }
  }
  // subclass of Point
  class Point3D extends Point {
    private int z;
    public Point3D(int _z) { ... }
    @Override
    public boolean equals(Object o1) {
       if(o1 != null && getClass() == o1.getClass()) {
         Point3D o = (Point3D) o1; //type casting
         return (super.equals(o1) && z==o.z);
       }
       else {
         return false;
       }
    }
  }
```

Today's Lecture

- Generic programming in Java
  - What?
  - Why?
  - How?
  - What not to do in generic programming?

Question

- By using any of the concepts taught till now in this course, how best can you store different types of objects?
    - 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 you with runtime errors

Approach 2

```
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
      if(generic.get(1) instanceof String) {
        String str = (String) generic.get(1); // OK
      }
    }
}
```

- We can use `instanceof` keyword to verify the type of object retrieved from `get()` function
  - Is this programmer friendly?
  - How many such “`if`” when you have several different types of objects in the list?

Approach 3

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

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

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

```
public class Main {
    public static void main(Sting args[]) {
        MyStringList strList = new MyStringList();
        MyIntList intList = new MyIntList();
        MyTypeXList typeXList = new MyTypeXList();

        strList.add(“hello”);
        intList.add(1);
        ...
    }
}
```

- We can create one class to hold one type of object
  - How many classes for N types of objects?
  - How many lines of code?
    - Is this programmer friendly?
      - NO !!

Solution: Generic Programming

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

Cup == Generic Container

Implementing generics

```
// a parameterized (generic) class
public class name<Type> {
or
public class name<Type1, Type2, ..., TypeN> {
```

- By putting the **Type** in < >, you are demanding that any client that constructs your object must supply a type parameter
  - You can require multiple type parameters separated by commas
- The rest of your class's code can refer to that type by name
- The type parameter is *instantiated* by the client. (e.g. E → String)

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 MyGenericList for different objects

```
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);
        ...
    }
}
```

A Generic Class with Multiple Fields

- Let’s create a class that could contain two different types of field, and type of both the fields are unknown

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));
        Pair<String, Integer> p = db.get(0);
    }
}
```

- Why this code isn't correct?
  - There is no compilation error although (Warning)

```
$ javac  Generic1.java

Note: Generic1.java uses unchecked or unsafe operations.

Note: Recompile with -Xlint:unchecked for details.
```

  - Database 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));
        Pair<String, Integer> p = db.get(0);
    }
}
```

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

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; }
}
```

```
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));
        Pair<String, Integer> p = db.get(0);
    }
}
```

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

Goals for Generic Programming

- Writing code that can be reused for objects of many different types
  - Programmer friendly

- For example, you don’t want to program separate classes to collect String and Integer objects

Behind the Scene: Generics are Implemented using Type Erasures

```
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);
    }
}

public class Main {
    public static void main(Sting args[]) {
        MyGenericList<String> strList = new
                               MyGenericList<String>();
        strList.add("hello");
        String str = strList.get(0);
    }
}
```

Compile Time

```
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 class Main {
    public static void main(Sting args[]) {
       MyGenericList strList = new MyGenericList();
       strList.add("hello");
       String str = (String) strList.get(0);
    }
}
```

- Compiler erases all parameter type information (type erasure)
- Compiler also ensures proper typecasting

Restrictions (1/6)

- Which of the following is correct?
  1. MyGenericList <double> var = new MyGenericList<Double>();

  2. MyGenericList <Double> var = new MyGenericList<Double>();

Type Parameters Cannot Be Instantiated with Primitive Types !
– No double, only Double

Restrictions (2/6)

```
public class MyGenericClass<T> {
    ......

    public void doSomething() {
        T my_var = new T(); // ERROR
    }

    public static void main(String[] arg){
        ......
    }
}
```

- Instantiating Type variable is not allowed
    - Compile time error
    - Type erasure removes the type information at runtime and hence its impossible to figure out the type at runtime

Restrictions (3/6)

```
public class MyGenericClass<T> {
    ......

    void doSomething(List<T> list) {
        if(list instanceof ArrayList<Integer>) {

        }
    }

    public static void main(String[] arg){
        .....
    }
}
```

- Cannot use casts or instanceof with parameterized types
    - Compile time error
    - Type erasure removes the type information at runtime and hence its impossible to figure out the type at runtime

Restrictions (4/6)

```
public class MyGenericClass<T> {
    ......
    private static T field;

    public static void main(String[] arg){
        MyGenericClass<Integer> c1 = new .....
        MyGenericClass<String> c2 = new .....
        MyGenericClass<Double> c3 = new .....
        // What is the type of “field” now ?
    }
}
```

- Cannot declare static fields whose types are Type parameter
  - Compilation error
  - If it was allowed then in the code shown here what will be the Type of “field” as it’s a static object hence shared by c1, c2 and c3

Restrictions (5/6)

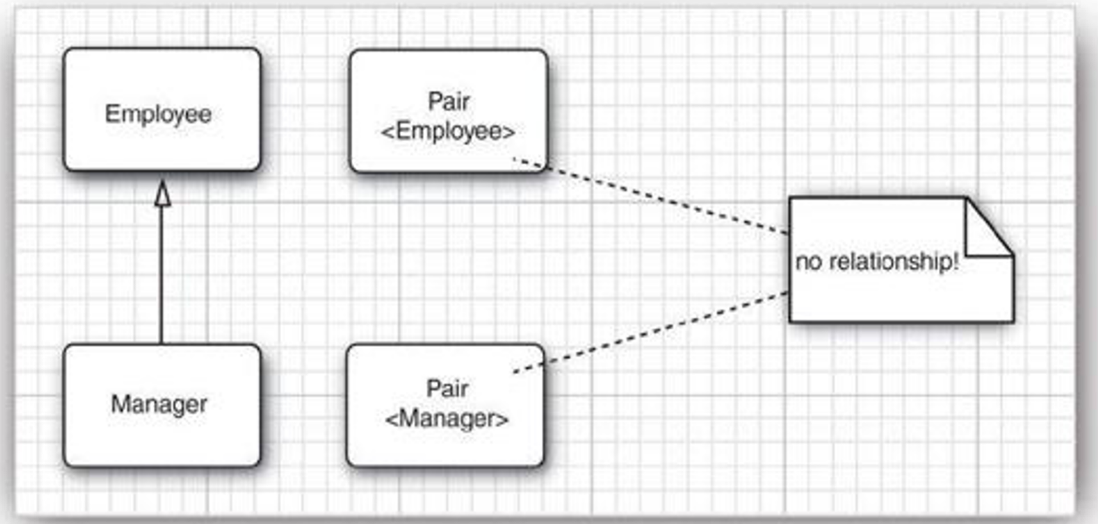

Image Source: Core Java, Volume-1

- Generic does not support sub-typing
  - If a class Employee is the superclass (parent) for a class Manager, then for a generic class Pair<T>, it does not mean Pair<Employee> also becomes the superclass (parent) for Pair<Manager>

Restrictions (6/6)

```
public class MyGenericClass<T> {
    ......

    public void doSomething() {
        T[] my_arr = new T[10]; // ERROR
    }

    public static void main(String[] arg){
        // ERROR
        MyGenericClass<String>[] str_array
                  = new MyGenericClass<String>[10];
    }
}
```

- Generic array creation is not allowed
  - Solution: create array of Object and typecast that array to generic type

Is there any Problem in Below Code?

```
public class Main {
    .....
   public static void print(ArrayList<Object> list){
        for(Object o: list)
            System.out.println(o);
    }
    public static void main(String[] arg){
        ArrayList<Integer> I = new
                      ArrayList<Integer>();
        I.add(1);
        I.add(2);
        ArrayList<String> S = new
                      ArrayList<String>();
        S.add(“Bob”);
        S.add(“Paul”);
        print(I);
        print(S);
    }
}
```

- The code won’t compile
- Although Object is superclass for Integer and String class, it does not mean that in the print method, `ArrayList<Object>` can hold `ArrayList<Integer>` or `ArrayList<String>`
    - Restriction-5 discussed in this lecture
- How to resolve this issue?
    - Except removing “<Object>” in print method

The WildCard to our Rescue !

The WildCard "?" to our Rescue !

```
public class Main {
    .....

   public static void print(ArrayList<?> list){
        for(Object o: list)
            System.out.println(o);
    }
    public static void main(String[] arg){
        ArrayList<Integer> I = new
                      ArrayList<Integer>();
        I.add(1);
        I.add(2);
        ArrayList<String> S = new
                      ArrayList<String>();
        S.add("Bob");
        S.add("Paul");
        print(I);
        print(S);
    }
}
```

- We just need **one** change in our code
- Simply use a wildcard character as type variable in the parameter `ArrayList` in print method

More Meaningful Example of Wildcard (1/2)

```
public class Main {
     .....
   static void print(ArrayList<? extends Car> 
list){
        .......
   }
   public static void main(String[] arg){
        .......
   }
}
```

- Upper bounded wildcard
    - Here the print method will only accept ArrayList of Car type or its subclass type

More Meaningful Example of Wildcard (2/2)

```
public class Main {
    .....
  static void print(ArrayList<? super Integer> list){
      .......
  }
  public static void main(String[] arg){
      .......
  }
}
```

- Lower bounded wildcard
  - Here the print method will only accept ArrayList of Integer or any Type that is supertype of Integer
    - Integer
    - Number
    - Object

Next Lecture

- Exception Handling
- **Quiz-2 on Friday in lab slot**
    - **4.15pm-4.35pm**