Generic Programming
- Part 1

Chapter 8, Core Java Volume I

Contents

- Why Generic Programming?
- Generic Classes
- Type Variables
- Advantages of Generics
- Generic Interfaces and Classes in Java Collection Framework
- Using a Generic Class
- Defining Generic Methods
- Bounds for Type Variables
- Type Erasure
- Cast Insertion
- Inheritance Rules for Generic Types
- Wildcard Types (Upper Bounded Wildcards)
- Supertype Bounds (Lower Bounded Wildcards)
- Unbounded Wildcards

Why Generic Programming?

- Generic programming = writing code that can be reused for objects of many different types. (a kind of polymorphism)
- Example: Pair of Something (in the same class)
  - Class PairString

```
public class PairString
{
   private String first;
   private Sting second;

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

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

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

Why Generic Programming?

- Example: Pair of Something
  - Class PairCard

```
public class PairCard
{
   private Card first;
   private Card second;

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

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

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

Why Generic Programming?

- Using Object class

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

   public Pair() { first = null; second = null; }
   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; }
}
```

Why Generic Programming?

- Drawbacks in using Object class
  - A cast is necessary whenever you retrieve a value:

```
Pair p1 = new Pair(“Kim”, “Lee”);  // pair of Objects

Card c1 = new Card(…);
Card c2 = new Card(…);

Pair p2 = new Pair(c1, c2);          // pair of Objects

String name = p1.getFirst();      // runtime error
Card c = p2.getSecond();          // runtime error

String name1 = (String) p1.getFirst();  // correct; cast is required
Card c = (Card) p2.getSecond();         // correct; cast is required
```

  - A pair object can contain different types: pair<String, Card>

```
Pair p = new Pair();
p.setFirst(new String(…));    // no error
p.setSecond(new Card(…));    // no error
```

Defining Generic Classes

- A *generic class* is a class with one or more *type variables* or type parameters.

```
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; }
}
```

type parameter
or type variables

Type Variables

- The T in public class Pair<T> is a type variable.
- The type variable is used throughout the class definition: fields, return value, local variables

  e.g. private T first;
- Instantiating the type like Pair<String>, substituting a type for the variable.
- You can think of the result as an ordinary class with these methods:

```
Pair<String>()
Pair<String>(String, String)
String getFirst()
String getSecond()
void setFirst(String)
void setSecond(String)
```

- A class can have multiple type variables:

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

The most commonly used type parameter names:

E - Element
K - Key
N - Number
T - Type
V - Value
S,U,V etc. - 2nd, 3rd, 4th types

Example: Testing Pair

```
public class PairTest1
{
   public static void main(String[] args)
   {
      String[] words = { "Mary", "had", "a", "little", "lamb" };
      Pair<String> mm = ArrayAlg.minmax(words);
      System.out.println("min = " + mm.getFirst());
      System.out.println("max = " + mm.getSecond());
   }
}
```

```
class ArrayAlg
{
   public static Pair<String> minmax(String[] a)
   {
      if (a == null || a.length == 0) return null;
      String min = a[0];
      String max = a[0];
      for (int i = 1; i < a.length; i++)
      {
         if (min.compareTo(a[i]) > 0) min = a[i];
         if (max.compareTo(a[i]) < 0) max = a[i];
      }
      return new Pair<>(min, max);
   }
}
```

Pair<String>
- type inference

Advantages of Generics

- No cast is required whenever you retrieve a value.

```
Pair<String> p1 = new Pair<String>(“Kim”, “Lee”);  // new Pair<>(…)

Card c1 = new Card(…);
Card c2 = new Card(…);

Pair<Card> p2 = new Pair<>(c1, c2);

String name = p1.getFirst();    // correct; cast no required
Card c = p2.getSecond();        // correct; cast no required
```

- Compiler checks argument types.

```
Pair<String> p = new Pair<>();
p.setFirst( new String(…) );
p.setSecond( new Card(…) );      // error; compiler checks argument types
```

Generic Interfaces and Classes in Java Collection Framework

- Collection<E>
- List<E>
- Queue<E>
- Set<E>
- ArrayList<E>
- LinkedList<E>
- Stack<E>
- PriorityQueue<E>
- HashSet<E>
- TreeSet<E>
- …

Using a Generic Classe in Java Collection Framework

```
public class ArrayListTest
{
   public static void main(String[] args)
   {
      ArrayList<Employee> staff = new ArrayList<>();

      staff.add(new Employee("Carl Cracker", 75000, 1987, 12, 15));
      staff.add(new Employee("Harry Hacker", 50000, 1989, 10, 1));
      staff.add(new Employee("Tony Tester", 40000, 1990, 3, 15));

      for (Employee e : staff)
         e.raiseSalary(5);

      for (Employee e : staff)
         System.out.println("name=" + e.getName() + ",salary=" + e.getSalary() + ",hireDay=" + e.getHireDay());
   }
}
```

Defining Generic Methods

- Generic method = method with type variables:

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

- When you call a generic method, the actual type comes before the method name:

```
String middle = ArrayAlg.<String>getMiddle(words);  // words is an array of String objects
Card middle = ArrayAlg.<Card>getMiddle(deck);         // deck is an array of Card objects
```

- Usually, the compiler infers the type from the argument types:

```
String middle = ArrayAlg.getMiddle(words);
```