Generics Programming

Chapter Topics

- Introduction to Generics
- Writing a Generic Class
- Passing Objects of a Generic Class to a Method
- Writing Generic Methods
- Constraining a Type Parameter in a Generic Class
- Inheritance and Generic Classes
- Defining Multiple Parameter Types
- Generics and Interfaces
- Erasure
- Restrictions of the Use of Generic Types

Generic Classes and Methods

- A generic class or method is one whose definition uses a placeholder for one or more of the types it works with.
- The placeholder is really a type parameter
- For a generic class, the actual type argument is specified when an object of the generic class is being instantiated.
- For a generic method, the compiler deduces the actual type argument from the type of data being passed to the method.

The ArrayList Class

The ArrayList class is generic: the definition of the class uses a type parameter for the type of the elements that will be stored.

ArrayList<String> specifies a version of the generic ArrayList class that can hold String elements only.

ArrayList<Integer> specifies a version of the generic ArrayList class that can hold Integer elements only.

Instantiation and Use of a Generic Class

ArrayList<String> is used as if it was the name of any non-generic class:

```
ArrayList<String> myList = new ArrayList<String>();
myList.add(“Java is fun”);
String str = myList.get(0);
```

A Generic Point Class

Consider a "point" as a pair of coordinates *x* and *y*, where *x* and *y* may be of any one type.

That is, if the type of *x* must always be the same as the type of *y*.

A Generic Point Class

```
class Point<T>                              // T represents a type parameter
{
  private T x, y;
  public Point(T x, T y)                     //  Constructor
  {
     set(x, y);
  }
  public void set(T x, T y)
  {
      this.x = x;   this.y = y;
  }
  T getX(){ return x;}
  T getY(){ return y;}
  public String toString()
  {
      return "(" +  x.toString() + ","  + y.toString() + ")";
  }
}
```

Using a Generic Class

```
public class Test
{
   public static void main(String [] s)
   {
      Point<String> strPoint = new Point<String>("Anna", "Banana");
      System.out.println(strPoint);
      Point<Number> pie = new Point<Number>(3.14, 2.71);
      System.out.println(pie);
   }
}
```

Program Output:

```
(Anna,Banana)
(3.14,2.71)
```

Reference Types and Generic Class Instantiation

Only reference types can be used to declare or instantiate a generic class.

```
ArrayList<Integer> myIntList = new ArrayList<Integer>;    // OK
ArrayList<int> myIntList = new ArrayList<int>;             // Error
```

int is not a reference type, so it cannot be used to declare or instantiate a generic class. You must use the corresponding wrapper class to instantiate a generic class with a primitive type argument.

Autoboxing

Autoboxing is the automatic conversion of a primitive type to the corresponding wrapper type when it is used in a context where a reference type is required.

```
// Autoboxing converts int to Integer
 Integer intObj = 35;
 // Autoboxing converts double to Number
 Point<Number> nPoint = new Point<Number>(3.14, 2.71);
```

Unboxing

Unboxing is the automatic unwrapping of a wrapper type to give the corresponding primitive type when the wrapper type is used in a context that requires a primitive type.

```
// Unboxing converts Integer to int
int i = new Integer(34);
// AutoBoxing converts doubles 3.14, 2.71 to Double
Point<Double> p = new Point<Double>(3.14, 2.71);
// p.getX() returns Double which is unboxed to double
double pi = p.getX();
```

Autoboxing, Unboxing, and Generics

Autoboxing and unboxing are useful with generics:

- Use wrapper types to instantiate generic classes that will work with primitive types

```
Point<Double> dPoint = new Point<Double>(3.14, 2.71);
```

- Take advantage of autoboxing to pass primitive types to generic methods:

```
dPoint.set(3.14, 2.71);
```

- Take advantage of unboxing to receive values returned from generic methods:

```
double pi = dPoint.getX();
```

Raw Types

You can create an instance of a generic class without specifying the actual type argument.

An object created in this manner is said to be of a raw type.

```
Point rawPoint = new Point("Anna", new Integer(26));
System.out.println(rawPoint);
```

Output:

(Anna, 26)

Raw Types and Casting

The Object type is used for unspecified types in raw types.

When using raw types, it is necessary for the programmer to keep track of types used and use casting:

```
Point rawPoint = new Point("Anna", new Integer(26));
System.out.println(rawPoint);
String name = (String)rawPoint.getX();      // Cast is needed
int age = (Integer)rawPoint.getY();          // Cast is needed
System.out.println(name);
System.out.println(age);
```

Commonly Used Type Parameters

Name	Usual Meaning
T	Used for a generic type.
S	Used for a generic type.
E	Used to represent generic type of an element in a collection.
K	Used to represent generic type of a key for a collection that maintains key/value pairs.
V	Used to represent generic type of a value for collection that maintains key/value pairs.

Generic Objects as Parameters

Consider a method that returns the square length of a Point object with numeric coordinates.

Square length of Point(3, 4) is 3*3 + 4*4 = 25

We can write the method:

```
static int sqLength(Point<Integer> p)
{
   int x = p.getX();
   int y = p.getY();
   return x*x + y*y;
}
```

The method is called as in

```
int i = sqLength(new Point<Integer>(3, 4));
```

Generics as Parameters

sqLength(Point<Integer> p) will not work for other numeric types and associated wrappers: for example, it will not work with Double.

We want a generic version of sqLength that works for all subclasses of the Number class.

Declaring the method parameter as Point<Number> works for Point<Number>, but not for any Point<T> where T is a subclass of Number:

```
static double sqLength(Point<Number> p)
{
  double x = p.getX().doubleValue();
  double y = p.getY().doubleValue();
  return x*x + y*y;
}
```

Works for:

```
Point<Number> p = new Point<Number>(3,4);
System.out.println(sqLength(p));
```

Does not work for:

```
Point<Integer> p = new Point<Integer>(3,4);
System.out.println(sqLength(p));            // Error
```

Wildcard Parameters

Generic type checking is very strict:

Point<Number> references cannot accept Point<T> objects unless T is Number.

A Point<Number> reference will not accept a Point<Integer> object, even though Integer is a subclass of Number.

The wildcard type symbol ? stands for any generic type:

Point<?> references will accept a Point<T> object for any type T.

Use of Wildcards

A version of sqLength using wildcards works for all subclasses of Number, but loses benefits of type checking, and requires casts.

```
static double sqLength(Point<?> p)
{
  Number n1 = (Number)p.getX();    // Needs cast to Number
  Number n2 = (Number)p.getY();
  double x = n1.doubleValue();
  double y = n2.doubleValue();
  return x*x + y*y;
}
```

Call as in

```
Point<Integer> p = new Point<Integer>(3,4);
System.out.println(sqLength(p));
```

Constraining Type Parameters

Benefits of type checking can be regained by constraining the wildcard type to be a subclass of a specified class:

```
Point <?> p1;                    // Unconstrained wildcard
Point <? extends Number> p2;  // Constrained wild card
```

p2 can accept a Point<T> object, where T is any type that extends Number.

Constraining Type Parameters

Casts no longer needed:

```
static double sqLength(Point<? extends Number> p)
{
    Number n1 = p.getX();
    Number n2 = p.getY();
    double x = n1.doubleValue();
    double y = n2.doubleValue();
    return x*x + y*y;
}
```

Call as in:

```
Point<Integer> p = new Point<Integer>(3,4);
System.out.println(sqLength(p));
```

Defining Type Parameters

The type parameter denoted by a wild card has no name.

If a name for a type parameter is needed or desired, it can be defined in a clause included in the method header.

The type definition clause goes just before the return type of the method.

Defining Type Parameters

Defining a type parameter is useful if you want to use the same type for more than one method parameter, or for a local variable, or for the return type.

Defining Type Parameters

Using the same type for several method parameters:

```
static <T extends Number>
void doSomething(Point<T> arg1, Point<T> arg2)
{

}
```

Using the name of the generic type for the return type of the method:

```
static <T extends Number>
Point<T> someFun(Point<T> arg1, Point<T> arg2)
{

}
```

Constraining Type Parameters

Type parameters can be constrained in Generic classes:

```
class Point<T extends Number>      //  T constrained to a subclass of Number
{
  private T x, y;
  public Point(T x, T y)  {  this.x = x;  this.y = y;  }
  double sqLength()
  {
     double x1 = x.doubleValue();
     double y1 = y.doubleValue();
     return x1*x1 + y1*y1;
  }
  T getX(){ return x;}
  T getY(){ return y;}
  public String toString()
  {
     return "(" +  x.toString() + ","  + y.toString() + ")";
  }
}
```

Type parameters can be constrained in Generic classes:

```
Point<Integer> p = new Point<Integer>(3,4);      //  Ok
System.out.println(p.sqLength());                //  Ok
Point<String> q = new Point<String>("Anna", "Banana");
                                                 //  Error, String is not a
                                                 // subclass of Number
```

Upper and Lower Bounds

The constraint <T extends Number > establishes Number as an upper bound for T. The constraint says T may be any subclass of Number.

A similar constraint <T super Number> establishes Number as a lower bound for T. The constraint says T may be any superclass of Number.

Inheritance and Generic Classes

Inheritance can be freely used with generic classes:

- a non-generic class may extend a generic class
- a generic class may extend a non-generic class
- a generic class may extend a generic class

A Generic Superclass

Consider this version of the generic Point class:

```
import java.awt.*;
class Point<T>
{
  protected T x, y;                    // protected x, y to allow inheritance
  public Point(T x, T y)
  {
    this.x = x;
    this.y = y;
  }
  T getX(){ return x;}
  T getY(){ return y;}
  public String toString()
  {
     return "(" +  x.toString() + ","  + y.toString() + ")";
  }
}
```

A Generic Subclass of a Generic Class

```
class ColoredPoint <T extends Number> extends Point<T>
{   private Color color;
    public ColoredPoint(T x, T y, Color c)
    {
        super(x, y);
        color = c;
    }
    public Color getColor() { return color;}    // Two subclass methods
    public double sqLength()
    {
        double x1 = x.doubleValue();
        double y1 = y.doubleValue();
        return x1*x1 + y1*y1;
    }
}
```

Examples of Use

```
public static void main(String [ ] s)
{
  // Can create subclass object
  ColoredPoint<Integer>
      p = new ColoredPoint<Integer>(3, 4,   Color.GREEN);
  System.out.println(p.sqLength());

  // Cannot create a ColoredPoint object parameterized with String
  ColoredPoint<String>
     q = new ColoredPoint<String>("Anna", "Banana", Color.GREEN);

  // Can create a Point object parameterized with String
  Point<String> q = new Point<String>("Anna", "Banana");
  System.out.println(q);
}
```

Defining Multiple Type Parameters

A generic class or method can have multiple type parameters:

```
class MyClass<S, T>
{

}
```

Multiple type parameters can be constrained:

```
class MyClass<S extends Number, T extends Date)
{

}
```

A Class with Multiple Type Parameters

```
class Pair<T, S>
{
   private T first;
   private S second;
   public Pair(T x, S y)
   {
      first = x; second = y;
   }
   public T getFirst(){ return first; }
   public S getSecond() {return second; }
}
```

Use of Multiple Type Parameters

Example of Instantiating and using an object of the Pair<T, S> generic class:

```
import java.util.Date;
public class Test
 {
  public  static void main(String [ ] args)
   {
      Pair<String, Date> p = new Pair<String, Date>("Joe", new Date());
      System.out.println(p.getFirst());
      System.out.println(p.getSecond());
   }
 }
```

Generics and Interfaces

- Interfaces, like classes, can be generic.
- An example of a generic interface in the class libraries is

```
public interface Comparable<T>
 {
   int compareTo(T o)
 }
```

This interface is implemented by classes that need to compare their objects according to some natural order.

The Comparable Interface

```
public interface Comparable<T>
{
    int compareTo(T o)
}
```

The compareTo method:

- returns a negative integer if the calling object is “less than” the other object.
- returns 0 if the calling object is “equal” to the other object.
- returns a positive integer if the calling object is “greater than” the other object.

Implementing the Comparable Interface

```
class Employee implements Comparable<Employee>
{
   private int rank;
   private String name;
   public int compareTo(Employee e)
   {
      return this.rank - e.rank;
   }
   public Employee(String n, int r)
   {
      rank = r;
      name = n;
   }
   public String toString()
   {
      return name + " : " + rank;
   }
}
```

Comparing Employee Objects

Sort two Employee objects by rank:

```
public class Test
{
   public  static void main(String [ ] args)
    {
       Employee bigShot = new Employee("Joe Manager", 10);
       Employee littleShot = new Employee("Homer Simpson", 1);
       if (bigShot.compareTo(littleShot) > 0)
       {
          System.out.println(bigShot);
          System.out.println(littleShot);
       }
       else
       {
          System.out.println(littleShot);
          System.out.println(bigShot);
       }
    }
}
```

Type Parameters Implementing Interfaces

A type parameter can be constrained to a type implementing an interface:

```
public static <T extends Comparable<T>>
T greatest(T arg1, T arg2)
{
    if (arg1.compareTo(arg2) > 0)
        return arg1;
    else
        return arg2;
}
```

Type Parameters Implementing Interfaces

The greatest method can be called as follows:

```
public  static void main(String [ ] args)
 {
    Employee bigShot = new Employee("Joe Manager", 10);
    Employee littleShot = new Employee("Homer Simpson", 1);
    Employee great = greatest(bigShot, littleShot);
    System.out.println(great);
 }
```

This avoids the need to pass objects as interfaces and then cast the return type from the interface back to the type of the object

Erasure

When processing generic code, the compiler replaces all generic types with the Object type, or with the constrained upper bound for generic type.

This process is known as erasure.

Effect of Erasure on Point<T>

```
class Point<T>
{
   private T x, y;
   public Point(T x1, T y1)
   {
       x = x1; y = y1;
   }
   public T getX() { return x;}
   public T getY() { return y;}
}
```

```
class Point
{
   private Object x, y;
   public Point(Object x1, Object y1)
   {
       x = x1; y = y1;
   }
   public Object getX() { return x;}
   public Object getY() { return y;}
}
```

Effect of Erasure on a Generic Method

```
static <E> void
displayArray(E [ ] array)
{
    for (E el : array)
   System.out.println(el);
}
```

```
static void
displayArray(Object [ ] array)
{
    for (Object el : array)
      System.out.println(el);
}
```

Erasure of Bounded Types

```
class Point<T extends Number>
 {
   private T x, y;
   public Point(T x1, T y1)
   {
      x = x1; y = y1;
   }
   public T getX() { return x;}
   public T getY() { return y;}
 }
```

```
class Point
 {
   private Number x, y;
   public
   Point( Number x1,  Number y1)
   {
      x = x1; y = y1;
   }
   public Number getX() { return x;}
   public Number getY() { return y;}
 }
```

Erasure of Bounded Type Parameter

```
<E extends Comparable<E>>
int search(E [ ] array, E val)
{
}
```

Becomes (after erasure):

```
int search(Comparable [ ] array, Comparable val)
{
}
```

Casting in Erasure

Once the generic types have been removed through erasure, the compiler introduces casting to make the raw types consistent with the actual types used.

Casting in Erasure

Assume the code

```
Integer x = new Integer(1);
Integer y = new Integer(2);
Point<Integer> myPoint = new Point<Integer>(x, y);
Integer tempX = myPoint.getX();
```

After erasure, the compiler will replace the last statement with:

```
Integer tempX = (Integer)myPoint.getX();
```

Restrictions on Instantiation of Type Parameters

A type parameter cannot be instantiated, so the following code will NOT compile:

```
<T> T myMethod(T x)
  {
     T myObj = new T();   // Error!
     return myObj;
  }
```

Erasure would replace T with Object. Thus the method would attempt to instantiate and return an Object. This is not the desired behavior.

Restriction on Generic Array Creation

You cannot create an array of a type that is an instance of a generic type. The following statement is an error:

Point<Integer> [ ] a = new Point<Integer>[10];

This is because the instantiation

new Point<Integer>[10]

needs to record the element type of the array. However, due to erasure, the type Point<Integer> does not exist, only a raw type named Point.

Legal Instantiation of Generic Arrays

```
class Point<T>
{
   T x, y;
   Point(T a, T b)
   {
      x = a; y = b;
   }
}
public class Test
{
  public  static void main(String [ ] args)
   {
      Point<? extends Number>[] a =  new Point[3];  // Use raw Type to instantiate
      a[0] = new Point<Integer>(3, 4);
      a[1] = new Point<Double>(3.14, 2.71);
      System.out.println(a[0].x);
      System.out.println(a[1].x);
   }
}
```

Restrictions on Static Fields

The type of a static field of a generic class may not be one of the class's type parameters. The following code is illegal.

```
class MyClass<T>
{
    static T sValue;    // Not permitted!
    T x;                // This is Ok.
    public MyClass( )
   {
      x = sValue;
   }
}
```

Because of erasure, there is only one copy of each static field.

Regardless of the number of instances of MyClass, the static field sValue can only have one type. But each instance of MyClass will have a different type for the type parameter T.

Restrictions on Static Methods

A static method may not have a local variable, or a parameter, whose type is one of the type parameters of the class.

```
class MyClass<T>
{
  static void doSomeThing()
  {
    T myValue;    // Not permitted!
  }
}
```

Again due to erasure, there can only be one actual type for myValue. But different instances of MyClass would need different actual types for T.