Generics

1.0 Introduction

Generics were introduced in J2SE 5.0. This enhancement to the type system allows a type or method to operate on objects of various types while providing compile-time type safety. It adds compile-time type safety to the Collections Framework and eliminates the need of type casting retrieving objects.

In a nutshell, generics enable *types* (classes and interfaces) to be parameters when defining classes, interfaces and methods. Much like the more familiar *formal parameters* used in method declarations, type parameters provide a way for you to re-use the same code with different inputs. The difference is that the inputs to formal parameters are values, while the inputs to type parameters are types.

At very high level, generics are nothing but parameterized types. Generics helps us to create a single class, which can be useful to operate on multiple data types. A class, interface or a method that operates on a parameterized type is called generics class, interface or method. Generics adds type safty. Remember that generics only works on objects, not primitive types.

Code that uses generics has many benefits over non-generic code:

- Stronger type checks at compile time.
  A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety. Fixing compile-time errors is easier than fixing runtime errors, which can be difficult to find.

- Elimination of casts.
  The following code snippet without generics requires casting:

  ```
  List list = new ArrayList();
  list.add("hello");
  String s = (String) list.get(0);
  ```

When re-written to use generics, the code does not require casting:

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);   // no cast
```

- Enabling programmers to implement generic algorithms.
  By using generics, programmers can implement generic algorithms that work on collections of different types, can be customized, and are type safe and easier to read.

2.0 Writing Generic Classes

2.1 Generic Types

A *generic type* is a generic class or interface that is parameterized over types. The following Box class will be modified to demonstrate the concept.

2.1.1 A Simple(non-generic) Box Class

Begin by examining a non-generic Box class that operates on objects of any type. It needs only to provide two methods: set, which adds an object to the box, and get, which retrieves it:

```
public class Box {
   private Object object;
   public void set(Object object) { this.object = object; }
   public Object get() { return object; }
}
```

Since its methods accept or return an Object, you are free to pass in whatever you want, provided that it is not one of the primitive types. There is no way to verify, at compile time, how the class is used. One part of the code may place an Integer in the box and expect to get Integers out of it, while another part of the code may mistakenly pass in a String, resulting in a run-time error.

2.1.2 A Generic Version of the Box Class

A *generic class* is defined with the following format:

```
class name<T1, T2, ..., Tn> { /* ... */ }
```

The type parameter section, delimited by angle brackets (<>), follows the class name. It specifies the *type parameters* (also called *type variables*) T1, T2, ..., and Tn.

To update the Box class to use generics, you create a *generic type declaration* by changing the code "public class Box" to "public class Box<T>". This introduces the type variable, T, that can be used anywhere inside the class.

With this change, the Box class becomes:

```
/**
 * Generic version of the Box class.
 * @param <T> the type of the value being boxed
 */
public class Box<T> {
   // T stands for "Type"
   private T t;
   public void set(T t) { this.t = t; }
   public T get() { return t; }
}
```

As you can see, all occurrences of Object are replaced by T. A type variable can be any **non-primitive** type you specify: any class type, any interface type, any array type, or even another type variable.

This same technique can be applied to create generic interfaces.

2.2 Type Parameter Naming Conventions

By convention, type parameter names are single, uppercase letters. This stands in sharp contrast to the variable naming conventions that you already know about, and with good reason: Without this convention, it would be difficult to tell the difference between a type variable and an ordinary class or interface name.

The most commonly used type parameter names are:

- *E - Element (used extensively by the Java Collections Framework)*
- *K - Key*
- *N - Number*
- *T - Type*
- *V - Value*
- *S,U,V etc. - 2nd, 3rd, 4th types*

2.3 Invoking and Instantiating a Generic Type

To reference the generic Box class from within your code, you must perform a *generic type invocation*, which replaces T with some concrete value, such as Integer:

Box<Integer> integerBox;

You can think of a generic type invocation as being similar to an ordinary method invocation, but instead of passing an argument to a method, you are passing a *type argument* — Integer in this case to the Box class itself.

Like any other variable declaration, this code does not actually create a new Box object. It simply declares that integerBox will hold a reference to a "Box of Integer", which is how Box<Integer> is read.

An invocation of a generic type is generally known as a *parameterized type*.

To instantiate this class, use the new keyword, as usual, but place <Integer> between the class name and the parenthesis:

Box<Integer> integerBox = new Box<Integer>();

2.3.1 The Diamond

In **Java SE 7 and later**, you can replace the type arguments required to invoke the constructor of a generic class with an empty set of type arguments (<>) as long as the compiler can determine, or infer, the type arguments from the context. This pair of angle brackets, <>, is informally called *the diamond*. For example, you can create an instance of Box<Integer> with the following statement:

Box<Integer> integerBox = new Box<>();

For more information on diamond notation and type inference, see Type Inference.

2.3.2 Multiple Type Parameters

As mentioned previously, a generic class can have multiple type parameters. For example, the generic OrderedPair class, which implements the generic Pair interface:

```
public interface Pair<K, V> {
   public K getKey();
   public V getValue();
}
public class OrderedPair<K, V> implements Pair<K, V> {
   private K key;
   private V value;
   public OrderedPair(K key, V value) {
         this.key = key;
         this.value = value;
   }
   public K getKey()       { return key; }
   public V getValue() { return value; }
}
```

The following statements create two instantiations of the OrderedPair class:

```
Pair<String, Integer> p1 = new OrderedPair<String, Integer>("Even", 8);

Pair<String, String>  p2 = new OrderedPair<String, String>("hello", "world");
```

The code, new OrderedPair<String, Integer>, instantiates K as a String and V as an Integer. Therefore, the parameter types of OrderedPair's constructor are String and Integer, respectively. Due to autoboxing, it is valid to pass a String and an int to the class.

To create a generic interface, follow the same conventions as for creating a generic class.

2.3.3 Parameterized Types

You can also substitute a type parameter (i.e., K or V) with a parameterized type (i.e., List<String>). For example, using the OrderedPair<K, V> example:

```
OrderedPair<String, Box<Integer>> p = new OrderedPair<>("primes", new Box<Integer>(...));
```

2.3.4 Raw Types

A *raw type* is the name of a generic class or interface without any type arguments. For example, given the generic Box class:

```
public class Box<T> {
   public void set(T t) { /* ... */ }
   // ...
}
```

To create a parameterized type of Box<T>, you supply an actual type argument for the formal type parameter T:

```
Box<Integer> intBox = new Box<>();
```

If the actual type argument is omitted, you create a raw type of Box<T>:

Box rawBox = new Box();

Therefore, Box is the raw type of the generic type Box<T>. However, a non-generic class or interface type is *not* a raw type.

Raw types show up in legacy code because lots of API classes (such as the Collections classes) were not generic prior to JDK 5.0. When using raw types, you essentially get pre-generics behavior — a Box gives you Objects. For backward compatibility, assigning a parameterized type to its raw type is allowed:

```
Box<String> stringBox = new Box<>();
Box rawBox = stringBox;            // OK
```

But if you assign a raw type to a parameterized type, you get a warning:

```
Box rawBox = new Box();           // rawBox is a raw type of Box<T>
Box<Integer> intBox = rawBox;     // warning: unchecked conversion
```

You also get a warning if you use a raw type to invoke generic methods defined in the corresponding generic type:

```
Box<String> stringBox = new Box<>();
Box rawBox = stringBox;
rawBox.set(8);  // warning: unchecked invocation to set(T)
```

The warning shows that raw types bypass generic type checks, deferring the catch of unsafe code to runtime. Therefore, you should avoid using raw types.

2.3.5 Unchecked Error Messages

As mentioned previously, when mixing legacy code with generic code, you may encounter warning messages similar to the following:

Note: Example.java uses unchecked or unsafe operations.

Note: Recompile with -Xlint:unchecked for details.

This can happen when using an older API that operates on raw types, as shown in the following example:

```
public class WarningDemo {
   public static void main(String[] args){
      Box<Integer> bi;
      bi = createBox();
   }
   static Box createBox(){
      return new Box();
   }
}
```

The term "unchecked" means that the compiler does not have enough type information to perform all type checks necessary to ensure type safety. The "unchecked" warning is disabled, by default, though the compiler gives a hint. To see all "unchecked" warnings, recompile with -Xlint:unchecked.

Recompiling the previous example with -Xlint:unchecked reveals the following additional information:

```
WarningDemo.java:4: warning: [unchecked] unchecked conversion
found   : Box
required: Box<java.lang.Integer>
      bi = createBox();
                   ^
1 warning
To completely disable unchecked warnings, use the -Xlint:-unchecked flag. The
@SuppressWarnings("unchecked") annotation suppresses unchecked warnings.
```

2.4 Writing Generic Methods

Generic methods are methods that introduce their own type parameters. This is similar to declaring a generic type, but the type parameter's scope is limited to the method where it is declared. Static and non-static generic methods are allowed, as well as generic class constructors.

The syntax for a generic method includes a type parameter, inside angle brackets, and appears before the method's return type. For static generic methods, the type parameter section must appear before the method's return type.

The Util class includes a generic method, compare, which compares two Pair objects:

```
public class Util {
   // Generic static method
   public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {
      return p1.getKey().equals(p2.getKey()) &&
           p1.getValue().equals(p2.getValue());
   }
}
public class Pair<K, V> {
   private K key;
   private V value;
   // Generic constructor
   public Pair(K key, V value) {
      this.key = key;
      this.value = value;
   }

   // Generic methods
   public void setKey(K key) { this.key = key; }
   public void setValue(V value) { this.value = value; }
   public K getKey()   { return key; }
```

```
    public V getValue() { return value; }
}
```

The complete syntax for invoking this method would be:

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = Util.<Integer, String>compare(p1, p2);
```

The type has been explicitly provided, as shown in bold. Generally, this can be left out and the compiler will infer the type that is needed:

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = Util.compare(p1, p2);
```

This feature, known as *type inference*, allows you to invoke a generic method as an ordinary method, without specifying a type between angle brackets.

2.4.1 Bounded Type Parameters

There may be times when you want to restrict the types that can be used as type arguments in a parameterized type. For example, a method that operates on numbers might only want to accept instances of Number or its subclasses. This is what *bounded type parameters* are for.

To declare a bounded type parameter, list the type parameter's name, followed by the extends keyword, followed by its *upper bound*, which in this example is Number. Note that, in this context, extends is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces).

```
public class Box<T> {
   private T t;
   public void set(T t) {
      this.t = t;
   }
   public T get() {
      return t;
   }
   public <U extends Number> void inspect(U u){
      System.out.println("T: " + t.getClass().getName());
      System.out.println("U: " + u.getClass().getName());
            }

   public static void main(String[] args) {
      Box<Integer> integerBox = new Box<Integer>();
      integerBox.set(new Integer(10));
      integerBox.inspect("some text"); // error: this is still String!
   }
}
```

By modifying the generic method to include this bounded type parameter, compilation will now fail, since our invocation of inspect still includes a String:

```
Box.java:21: <U>inspect(U) in Box<java.lang.Integer> cannot   be applied to (java.lang.String)
                integerBox.inspect("10");
```

```
                    ^
1 error
```

In addition to limiting the types you can use to instantiate a generic type, bounded type parameters allow you to invoke methods defined in the bounds:

```
public class NaturalNumber<T extends Integer> {
   private T n;
   public NaturalNumber(T n)  { this.n = n; }
   public boolean isEven() {
      return n.intValue() % 2 == 0;
   }
//..
}
```

The isEven method invokes the intValue method defined in the Integer class through n.

2.4.2 Multiple Bounds

The preceding example illustrates the use of a type parameter with a single bound, but a type parameter can have *multiple bounds*:

```
<T extends B1 & B2 & B3>
```

A type variable with multiple bounds is a subtype of all the types listed in the bound. If one of the bounds is a class, it must be specified first. For example:

```
Class A { /* ... */ }
interface B { /* ... */ }
interface C { /* ... */ }
class D <T extends A & B & C> { /* ... */ }
If bound A is not specified first, you get a compile-time error:
class D <T extends B & A & C> { /* ... */ }  // compile-time error
```

2.5 Restrictions on Generics

To use Java generics effectively, you must consider the following restrictions:

- Cannot Instantiate Generic Types with Primitive Types
- Cannot Create Instances of Type Parameters
- Cannot Declare Static Fields Whose Types are Type Parameters
- Cannot Use Casts or instanceof With Parameterized Types
- Cannot Create Arrays of Parameterized Types
- Cannot Create, Catch, or Throw Objects of Parameterized Types
- Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type

Cannot Instantiate Generic Types with Primitive Types:

Consider the following parameterized type:

```
class Pair<K, V> {
   private K key;
```

```
    private V value;
    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }
}
```

When creating a Pair object, you cannot substitute a primitive type for the type parameter K or V:

```
Pair<int, char> p = new Pair<>(8, 'a');  // compile-time error
```

You can substitute only non-primitive types for the type parameters K and V:

```
Pair<Integer, Character> p = new Pair<>(8, 'a');
```

Note that the Java compiler autoboxes 8 to Integer.valueOf(8) and 'a' to Character('a'):

```
Pair<Integer, Character> p = new Pair<>(Integer.valueOf(8), new Character('a'));
```

Cannot Create Instances of Type Parameters:

You cannot create an instance of a type parameter. For example, the following code causes a compile-time error:

```
public static <E> void append(List<E> list) {
    E elem = new E();  // compile-time error
    list.add(elem);
}
```

As a workaround, you can create an object of a type parameter through reflection:

```
public static <E> void append(List<E> list, Class<E> cls) throws Exception {
    E elem = cls.newInstance();   // OK
    list.add(elem);
}
```

You can invoke the append method as follows:

```
List<String> ls = new ArrayList<>();
append(ls, String.class);
```

Cannot Declare Static Fields Whose Types are Type Parameters:

A class's static field is a class-level variable shared by all non-static objects of the class. Hence, static fields of type parameters are not allowed. Consider the following class:

```
public class MobileDevice<T> {
    private static T os;

    // ...
}
```

If static fields of type parameters were allowed, then the following code would be confused:

```
MobileDevice<Smartphone> phone = new MobileDevice<>();
MobileDevice<Pager> pager = new MobileDevice<>();
MobileDevice<TabletPC> pc = new MobileDevice<>();
```

Because the static field os is shared by phone, pager, and pc, what is the actual type of os? It cannot be Smartphone, Pager, and TabletPC at the same time. You cannot, therefore, create static fields of type parameters.

Cannot Use Casts or instanceof with Parameterized Types:

Because the Java compiler erases all type parameters in generic code, you cannot verify which parameterized type for a generic type is being used at runtime:

```
public static <E> void rtti(List<E> list) {
   if (list instanceof ArrayList<Integer>) {  // compile-time error
      // ...
   }
}
```

The set of parameterized types passed to the rtti method is:

S = { ArrayList<Integer>, ArrayList<String> LinkedList<Character>, ... }

The runtime does not keep track of type parameters, so it cannot tell the difference between an ArrayList<Integer> and an ArrayList<String>. The most you can do is to use an unbounded wildcard to verify that the list is an ArrayList:

```
public static void rtti(List<?> list) {
   if (list instanceof ArrayList<?>) {  // OK; instanceof requires a reifiable type
      // ...
   }
}
```

Typically, you cannot cast to a parameterized type unless it is parameterized by unbounded wildcards. For example:

```
List<Integer> li = new ArrayList<>();
List<Number>  ln = (List<Number>) li;  // compile-time error
```

However, in some cases the compiler knows that a type parameter is always valid and allows the cast. For example:

```
List<String> l1 = ...;
ArrayList<String> l2 = (ArrayList<String>)l1;  // OK
```

Cannot Create Arrays of Parameterized Types:

You cannot create arrays of parameterized types. For example, the following code does not compile:

```
List<Integer>[] arrayOfLists = new List<Integer>[2];  // compile-time error
```

The following code illustrates what happens when different types are inserted into an array:

```
Object[] strings = new String[2];
strings[0] = "hi";   // OK
strings[1] = 100;    // An ArrayStoreException is thrown.
```

If you try the same thing with a generic list, there would be a problem:

```
Object[] stringLists = new List<String>[];  // compiler error, but pretend it's allowed
stringLists[0] = new ArrayList<String>();   // OK
stringLists[1] = new ArrayList<Integer>();  // An ArrayStoreException should be thrown,
                                            // but the runtime can't detect it.
```

If arrays of parameterized lists were allowed, the previous code would fail to throw the desired ArrayStoreException.

Cannot Create, Catch, or Throw Objects of Parameterized Types:
A generic class cannot extend the Throwable class directly or indirectly. For example, the following classes will not compile:

```
// Extends Throwable indirectly
class MathException<T> extends Exception { /* ... */ }    // compile-time error
// Extends Throwable directly
class QueueFullException<T> extends Throwable { /* ... */ // compile-time error
```

A method cannot catch an instance of a type parameter:

```
public static <T extends Exception, J> void execute(List<J> jobs) {
   try {
      for (J job : jobs)
         // ...
   } catch (T e) {   // compile-time error
      // ...
   }
}
```

You can, however, use a type parameter in a throws clause:

```
class Parser<T extends Exception> {
   public void parse(File file) throws T {     // OK
      // ...
   }
}
```

Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type:
A class cannot have two overloaded methods that will have the same signature after type erasure.

```
public class Example {
   public void print(Set<String> strSet) { }
   public void print(Set<Integer> intSet) { }
}
```

The overloads would all share the same classfile representation and will generate a compile-time error.

Using Generics in Programming
Generics at Class-level
As discussed earlier generics can be used at both class-level as well as at method-level. The example below shows how to write a generic class.

The Box class is defined as a generic class. The type of box created is only identified at the time of the creation of the object of the box. The methods and parameters inside the Box class are also of generic type allowing to create a n object of the Box of any of the different types.

```
public  class Box<T>{
    private T b;

    Box (T b){
        this.b=b;
    }

    public T getBox(){
        return b;
    }

    @Override
    public String toString(){
    return b.getClass().getName();
    }

    public static void main(String args[]){
        Box<String> b1 = new Box<String>("Java Generics");
        System.out.println("Type: " +b1.toString() + " " +b1.getBox());

        Box<Integer> b2 = new Box<Integer>(1);
        System.out.println("Type: " +b2.toString() + " " + b2.getBox());
    }
}
```

The output of the program looks like this:

```
Type: java.lang.String Java Generics

Type: java.lang.Integer 1
```

What happens in case you want to access a group of objects that derive from same super class, meaning extending same super class? You can restrict the generics type parameter to a certain group of objects which extends the same super class. You can achieve this my specifying extends <super-class> at class-level definition.

Here is the example of how to implement bounded types (extending super classes) using generics

```
public class ExtendSuperClassExample<T extends A> {

    private T t;

    public ExtendSuperClassExample (T t){
            this.t = t;
        }
    public void showClassName(){
            this.t.printClass();
        }

    public static void main(String[] args) {
        ExtendSuperClassExample<A> ea= new ExtendSuperClassExample<A>(new A());
        ea.showClassName();

        ExtendSuperClassExample<B> eb= new ExtendSuperClassExample<B>(new B());
        eb.showClassName();
    }
}

class A {
    public void printClass(){
        System.out.println("I am in Super Class A");
    }
}

class B extends A{
    public void printClass(){
        System.out.println("I am in Sub Class B");
    }
}
```

The output of the program looks like this:

```
I am in Super Class A

I am in Sub Class B
```

Generics at method-level:

The example below shows how to write a generic method. The method findMax(T x, T y, T z) will find the maximum of three values supplied to it. The method is generalized to accept as parameters objects or values of any java types of which it will find the maximum of the three values supplied.

```
public class MaximumTest
{
   // determines the largest of three Comparable objects
   public static <T extends Comparable<T>> T findMax(T x, T y, T z)
   {
      T max = x; // assume x is initially the largest
      if ( y.compareTo( max ) > 0 ){
         max = y; // y is the largest so far
      }
      if ( z.compareTo( max ) > 0 ){
         max = z; // z is the largest now
      }
      return max; // returns the largest object
   }

   public static void main( String args[] )
   {
      System.out.printf( "Max of %d, %d and %d is %d\n\n",
                  3, 4, 5, findMax( 3, 4, 5 ) );

      System.out.printf( "Max of %.1f,%.1f and %.1f is %.1f\n\n",
                  6.6, 8.8, 7.7, findMax( 6.6, 8.8, 7.7 ) );

      System.out.printf( "Max of %s, %s and %s is %s\n","pear",
         "apple", "orange", findMax( "pear", "apple", "orange" ) );
   }
}
```

The output of the program looks like this:

```
Max of 3, 4 and 5 is 5

Maxm of 6.6,8.8 and 7.7 is 8.8

Max of pear, apple and orange is pear
```

Using Generics in Collections

Note that all the core collection interfaces are generic. For example, this is the declaration of the Collection interface.

```
public interface Collection<E>...
```

The <E> syntax tells you that the interface is generic. Meaning at the time of declaring a collection instance you can and must specify the type(s) of object(s) it can (or cannot) contain. Specifying the

type allows the compiler to verify (at compile-time) that the type of object you add into the collection is correct, thus reducing errors at runtime.

Java Collection objects can store data elements that are of one single type or of multiple /mixed data types in the same collection object. The return type of the data element that is retrieved from a collection is Object. However when retrieving you will need to type-cast the object into its actual data type. While doing so there can be programmatic errors as in type-casting to a wrong type which will result in a run-time error.

By using generics it is possible to tell the compiler particularly what type of data a Collection object is going to store. This allows the compiler to check for any unintentional errors like adding the wrong type of data to the collection during compile-time itself. It also avoids the need to type-cast the retrieved objects. Generics add compile-time type safety to the Collections Framework and eliminates the need of type casting retrieving objects.

The example below demonstrates how to use generics with Collections.

```
import java.util.ArrayList;
import java.util.Iterator;

public class CollectionsTest<T>{
    ArrayList<T> al ; //ArrayList of generic type
    CollectionsTest(ArrayList<T> t){
        this.al=t;
    }
    public void setList(T t){
        al.add(t);
    }
    public void printList(){
        // create a generic iterator, the type T will be replaced by the actual type
        // passed to constructor at run-time
        Iterator<T> itr=al.iterator();
        while(itr.hasNext()){
            System.out.println(itr.next());
        }
    }
    public static void main(String args[]){
        // Create an ArrayList that stores strings
    CollectionsTest<String> ct1 = new CollectionsTest<String>(new ArrayList<String>());
    ct1.setList("abc");
    ct1.setList("xyz");
    ct1.printList();
    // Create an ArrayList that stores integers
    CollectionsTest<Integer> ct2 = new CollectionsTest<Integer>(new ArrayList<Integer>());
    ct2.setList(1);
    ct2.setList(2);
    ct2.printList();
    }
}
```

The output of the program will look like this:

```
abc

xyz
```

```
1
```

```
2
```

References

- http://docs.oracle.com/javase/tutorial/java/generics/index.html
- http://www.java2novice.com/java-generics/
- http://www.tutorialspoint.com/java/java_generics.htm