Java Brand Generics – Light Version

Advanced Topics in Java

Khalid Azim Mughal
khalid@ii.uib.no
http://www.ii.uib.no/~khalid/

Version date: 2007-02-03

Overview

- Introduction to Generics
- Basic Java Generics
- Increasing Expressive Power with Wildcard Type Parameters
- Generic Interfaces
- Generic Methods
- Other Implications and Restrictions regarding Java Generics

The Role of Generics in Programming Languages

- *Generics*[1] allow definition and implementation of *generic abstractions*.
  - The *generic type* is parameterized by one or more *formal type parameters*.
  - The *actual type parameters* are supplied when the generic type is *instantiated*.
  - A *parameterized type* thus represents a *set of types* depending on the *actual type parameters* supplied.
- Recurring theme in the evolution of programming languages
  - For example: Ada, Clu, C++, C#, Eiffel, and now Java

"One List to Rule Them All!"

1. a.k.a. Genericity, Parametric polymorphism

Java Brand of Generics

- Goal:

  Provide extra type information at compile time to avoid verbosity and the necessity of explicit type checking and casting at runtime.
- Compile-time Java Generics (JG):

  Reference types (classes, interfaces and array types) and methods can be parameterized with type information.

  Generics implemented as compiler transformations, with insignificant impact on the JVM.
- Benefits:

  Increased language expressiveness with improved type safety

BASIC JAVA GENERICS

- The Need for Generics
- Generic Types
- Parameterized Types
- Canonical usage with Collections

Canonical Problem: Collections w/o Generics

- A collection without generics can hold references to objects of any type.

```
List wordList = new ArrayList();    // Using a generic reference for the list.
wordList.add("two zero zero five"); // Can add any object.
wordList.add(new Integer(2004));
//...
Object element = wordList.get(0);
//...
if (element instanceof String) {   // Runtime check to avoid ClassCastException
  String strInt = (String) element;// Cast required.
  //...
}
```

- Inheritance allows the implementation of the class to be specific, but its use to be generic.
  – An `ArrayList` is a *specific implementation* of the `List` interface, *usage* of the class `ArrayList` is *generic* with regard to any object.

Collections w/ Generics

- Using *parameterized types*:

```
List<String> wordList = new ArrayList<String>(); // Using a specific type.
wordList.add(C);              // Can add only strings
wordList.add(new Integer(2004));              // Compile-time error!
//...
String element = wordList.get(0);             // Only strings as elements
//...
```

- *No runtime check or explicit cast necessary!*
- *Generic types* allow the implementation of class to be generic, but its use to be specific.
  - The generic type `ArrayList<E>` is a *generic implementation* of the `List<E>` interface, *usage* of the parameterized type `ArrayList<String>` is *specific*, as it constrains the generic type `ArrayList<E>` to strings.

```
// Implementation in the java.util package
public interface List<E> extends Collection<E> { ... }
public class ArrayList<E> extends AbstractList<E> { ... }
```


Generic Types and Parameterized Types

- A *generic type* is a reference type that defines a set of *formal type parameters* or *type variables* (E_1, E_2...E_n) that must be provided for its invocation.

```
public class ArrayList<E> extends AbstractList<E> { ... } // (1) Declaration
```

  - The *(formal) type parameter* is an *unqualified identifier.*
  - The type parameter `E` can be used (pretty much) as any other type in the class, although a type parameter cannot be used to create a new instance.
  - A generic type without its formal type parameters is called a *raw type*.
    - `ArrayList` is the raw type of the generic type `ArrayList<E>`.
- An *invocation* or *instantiation* (usually called a *parameterized type*) is a specific usage of a generic type where the formal type parameters are replaced by *actual type parameters*.

```
ArrayList<String> mylist = new ArrayList<String>();       // (2) Invocation
```

- Methods can be called on objects of generic types, no extra syntax is required:

```
myList.add("two zero zero five");
```

We can declare references of generic types, instantiate generic classes, and call methods on the objects.

General Remarks on Java Generics

- The compiler ensures that the parameterized type is used correctly so that errors are caught at *compile time*, and not at runtime.
- Generics are implemented in the compiler; *no* generic type info is available at runtime.
- A parameterized type does *not* create a new class.
  - The invocations share the generic type.
- Invocation of generic types is restricted to reference types (excluding array creation and enumerations), and primitive types are *not* permitted as type parameters.

Example of legacy code: LegacySeq

- Any object can be maintained in a sequence, the client must do the bookkeeping.

```
public class LegacySeq {
  private Object    element; // Data
  private LegacySeq tail;    // Rest of the sequence
  LegacySeq(Object element, LegacySeq tail) {
    this.element = element;
    this.tail = tail;
  }
  public void   setElement(Object obj) { element = obj; }
  public Object getElement()           { return element; }
  public void   setTail(LegacySeq tail) { this.tail = tail; }
  public LegacySeq getTail()            { return this.tail; }
  // ...
}
// Client code
LegacySeq intSeq = new LegacySeq(32, new LegacySeq(16, null));
intSeq.setElement(8.5);                      // Any object can be added.
Integer iRef = (Integer) intSeq.getElement(); // ClassCastException at runtime.
```

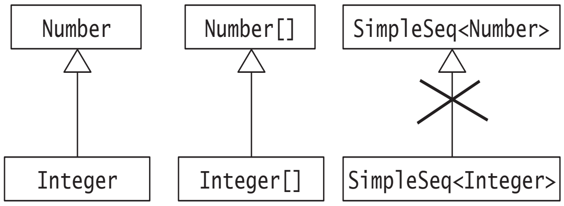
Example of (naive) generic code: SimpleSeq

```
public class SimpleSeq<T> {
  private T            element; // Data
  private SimpleSeq<T> tail;    // Rest of the sequence
  SimpleSeq(T element, SimpleSeq<T> tail) {
      this.element = element;
      this.tail = tail;
  }
  public void  setElement(T obj) { element = obj; }
  public T     getElement()      { return element; }
  public void  setTail(SimpleSeq<T> tail) { this.tail = tail; }
  public SimpleSeq<T> getTail()          { return this.tail; }
  //...
}
// Client code: declaring references and instantiating objects of generic classes
  SimpleSeq<Integer> intSeq    = new SimpleSeq<Integer>(10, null);   // (1)
  SimpleSeq<Double>  doubleSeq = new SimpleSeq<Double>(20.5,null);   // (2)
  SimpleSeq<Number>  numSeqA = new SimpleSeq<Number>(30.5, null);    // (3)
  SimpleSeq<Number>  numSeqB = new SimpleSeq<Number>(30.5, numSeqA); // (3)
//SimpleSeq<Number>  numSeqC = new SimpleSeq<Number>(40.5, intSeq);  // (4) Huh?
//SimpleSeq<Number>  numSeqD = doubleSeq;                            // (5) Huh?
```


Example of (naive) generic code: SimpleSeq (cont.)

- The generic type `SimpleSeq`<T> allows only a sequence of a *specific type* to be maintained.
- The scope of the type parameter `T` is the declaration of the generic type.
  – All occurrences of the `Object` class in the `LegacySeq` class have been replaced by the type parameter `T` in the `SimpleSeq` class.
  – The usage of the class name `SimpleSeq` is parameterized by the type parameter `T` in the class declaration.
- The type parameter `T` binds to the actual type parameter specified in the invocation of the generic type.
- Note that in the implementation of the generic type `SimpleSeq<T>`, we never invoke methods on objects of the type parameter `T`.
  – This is not always the case in implementing generic types.

No Subtype Covariance for "Pure" Parameterized Types



- What is the problem?

```
SimpleSeq<Integer> intSeq = new SimpleSeq<Integer>(64, null);
SimpleSeq<Number> numSeq  = intSeq;   // (1) DISALLOWED: compile-time error
numSeq.setElement(3.14);              // (2) No runtime type info available
Integer iRef = intSeq.getElement();   // (3) Runtime type error
```


INCREASING EXPRESSIVE POWER WITH WILDCARD TYPE PARAMETERS

- Defining Variant Parametric Types with Wildcards
  – Type Parameter Upper and Lower Bounds
- Understanding Subtype-Supertype Relationships involving Generic Types
- Using Wildcards

Running example classes:

```
class Seq<T> { ... }
class SafeSeq<T> extends Seq<T> { ... }
```

Type Specification Overview

Name	Syntax	Semantics	Description
Subtype Covariance	<? extends T>	Any subtype of T	Wildcard with upper bound
Subtype Contravariance	<? super T>	Any supertype of T	Wildcard with lower bound
Subtype Bivariance	<?>	Any type	Unbounded Wildcard
Subtype Invariance	<T>	Only type T	"Pure" generics

Subtype Covariance: <? extends T>

- The *subtype covariance* relation is represented by the following *bounded wildcard*:

  <? extends T>

  – ? denotes an unknown type.

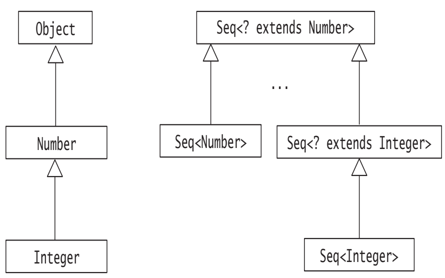
  – T is called the *upper bound*.

  – The wildcard <? extends T> means that any *subtype* of T (or T itself) is acceptable as an actual type parameter in the wildcard invocation.

  Example:

  – The wildcard <? extends Number> denotes a *family of subtypes* of Number.

  – The invocation Seq<? extends Number> denotes *a set of invocations* of Seq for types that are *subtypes* of Number.



Subtype Contravariance: <? super T>

- The *subtype contravariance* relation is represented by the following *bounded wildcard*:

  <? super T>

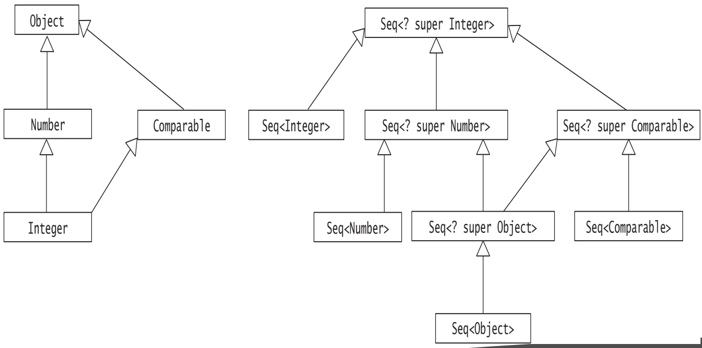
  – T is called the *lower bound*.

  – The wildcard <? super T> means that any *supertype* of T (or T itself) is acceptable as an actual type parameter in the wildcard invocation.

  Example:

  – The wildcard <? super Integer> denotes a *family of supertypes* of Integer.

  – The wildcard invocation Seq<? super Integer> denotes *a set of invocations* of Seq for types that are *supertypes* of Integer.



Subtype Bivariance: <?>

- The *subtype bivariance* relation is represented by the *unbounded wildcard*, **<?>**.

  – By definition, the bivariance relation is covariant and contravariant.

  – It denotes the family of all types.

  Example:

  – The *wildcard invocation* Seq<?> denotes *the set of invocations* of Seq for any type, i.e. denotes a Seq of anything.

Subtype Invariance: <T>

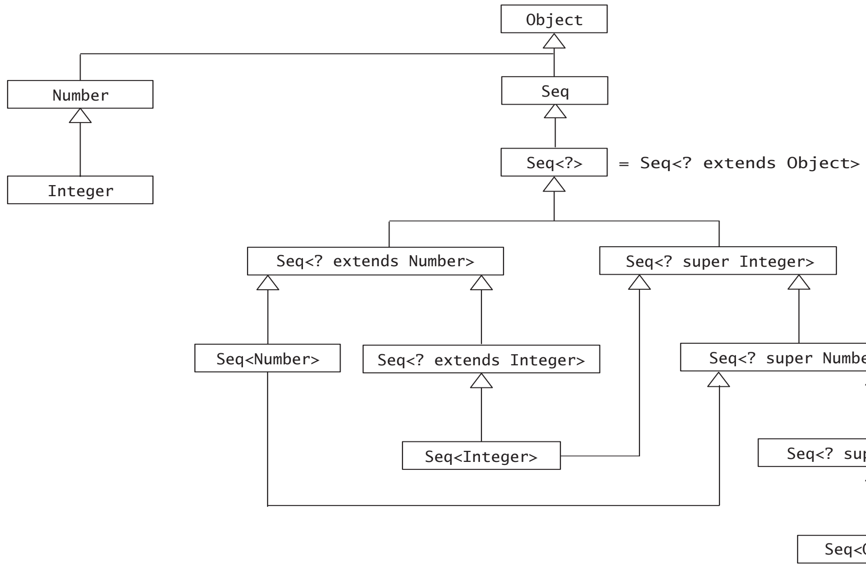
- The *subtype invariance* relation is represented by **<T>**, there T is a specific type.

  – The notation <T> means that only T is acceptable as an actual type parameter in the invocation.

  Example: "Pure" Generics

  – The *invocation* Seq<Integer> denotes a set containing only the instantiation of Seq for the specified type parameter.

Variant Parametric Type Hierarchy (partial view)



Generic Class **Seq<T>**

```
public class Seq<T> {
  private T                element; // Data
  private Seq<? extends T> tail;    // Rest of the sequence
  Seq(T element, Seq<? extends T> tail) {
      this.element = element;
      this.tail = tail;
  }
  public void    setElement(T obj) { element = obj; }
  public T       getElement()      { return element; }
  public void    setTail(Seq<? extends T> tail) { this.tail = tail; }
  public Seq<? extends T> getTail()             { return this.tail; }
  public void sort(Comparator<? super T> comp)  {/*...*/}
  public String toString() {
    return this.element.toString() + (this.tail == null? "" : ", " + this.tail);
  }
  // ...
}
```

Reference Declarations With Wildcards

- A wildcard can be used for declaration of references, but *not* for creating objects.
- The following code does not compile because the type parameter is of unknown type in the wildcard invocation of the generic type.

```
Seq<? extends Number> numSeq = new Seq<?>(2006, null); // Compile-time error
```

  – The type of the object cannot be deduced.
- Such a reference can refer to an object whose type is a parameterized type in the set of invocations of the generic type denoted by the wildcard invocation.
  – i.e. *assignment compatibility* is according to the variant parametric type hierarchy of the wildcard instantiations.

```
  Seq<Integer> intSeq    = new Seq<Integer>(10, null);
  Seq<Number>  numSeqC   = new Seq<Number>(50.5, intSeq);
//Seq<Number>  numSeqD   = intSeq;                 // Compile-time error
  Seq<? extends Number>  numSeqE = intSeq;
  Seq<? extends Integer> numSeqF = null;
  Seq<? super Integer> numSeqG = new Seq<Integer>(2004, null);
```


Flexibility with Wildcard Type Parameters

- Consider the following two implementations of the `sort()` method for the generic class `Seq<T>`:

```
public void sort(Comparator<T> comp) {/*...*/ }          // Alt. (1)

public void sort(Comparator<? super T> comp) {/*...*/ }  // Alt. (2)
```

- Client code:

```
Seq<String> strSeq = new Seq<String>("aha", new Seq<String>("aho", null));
strSeq.sort(String.CASE_INSENSITIVE_ORDER); // Comparator<String> is ok for (1)
                                            //   and (2).
strSeq.sort(new Comparator<Object>() {      // Comparator<Object> is not ok
  public int compare(Object o1, Object o2)  //   for (1) and but is ok for (2).
    {return ...;}
});
```

- Alt. 2 gives greater flexibility in choice of comparator.

Type Parameters with Multiple Bounds

- A type parameter T can have *multiple bounds*:

```
T extends MyClass & Comparable<T> & Serializable
```

- Example of usage:

```
class SomeClass<T extends MyClass & Comparable<T> & Serializable> { /*...*/ }
```

- If the raw type of a bound is a superclass, then it can only be specified as the first bound and there can only be one such bound (as a subclass can only extend one immediate superclass).
- A type parameter T having multiple bounds is a subtype of all of the types denoted by the individual bounds.
- The raw type of an individual bound cannot be used with different arguments:

```
E extends Object & Comparable<E> & Serializable & Comparable<String> // Not OK.
```

- The first bound is used as the *type erasure* for the type parameter T (*see discussion on type erasures*).
- Note the use of the type parameter T in the bound `Comparable<T>`.
  - In the invocation `SomeClass<MyType>`, the compiler ensures that `MyType` also implements `Comparable<MyType>`.

Generic Interfaces

- Specification of a generic interface is analogous to that of a generic class:

```
public interface Comparable<E> {
  int compareTo(E thingy);
}
```

- Invocation of a generic interface is analogous to that of a generic class, but it is done in conjunction with a class implementing the interface:

```
// Non-generic class implements generic interface
class Node implements Comparable<Node> {
  // ...
  int compareTo(Node thingy) { ... }
  // ...
}

// Generic class implements generic interface
public class ArrayList<E>
             extends AbstractList<E>
             implements List<E>, RandomAccess, Cloneable, Serializable {
  //...
}
```

GENERIC METHODS

- Declaring Generic Methods
- Calling Generic Methods
- Limitations on Generic Methods and Wild Card Usage

Declaring Generic Methods

- A generic method (a.k.a. *polymorphic method*) is implemented like an ordinary method, except that a type parameter is specified immediately preceding the return type.
- Examples: Two attempts at writing a generic method that returns the "maximum" of two objects.

```
  static <T> T max1(T obj1, T obj2) { // Note the type parameter.
    T result = obj1;
    if (obj1.compareTo(obj2) < 0)  // Compiler-time error: method compareTo(T) not
found
      result = obj2;
    return result;
  }
  static <T extends Comparable<T>> T max2(T obj1, T obj2) {
    T result = obj1;
    if (obj1.compareTo(obj2) < 0)  // OK
      result = obj2;
    return result;
  }
```

- A generic method need not be in a generic class.
- If declared in a generic class, a generic method can also use the type parameters of the class.

Calling Generic Methods

- A generic method can be called like an ordinary method, without any actual type parameter.
  – The parameter type (and the return type) is inferred from the type of the actual parameters.

```
int maxInt = max2(12,23);              // OK
String maxStr = max2("aha", "madonna"); // OK
Number numResult= max2(new Integer(12), new Double(23.0)); // Compile-time error
```

- Calling generic methods whose parameters have parameterized types.

```
// Generic Methods:
  static <T> void merge1 (Seq<T> s1, Seq<T> s2) { /*...*/ };
  static <T> void merge2 (Seq<T> s1, Seq<? extends T> s2) { /*...*/ };
  static <T> void merge3 (Seq<T> s1, Seq<? super T> s2) { /*...*/ };
// Client code:
    Seq<Number> numSeq = new Seq<Number>(31.4, null);
    Seq<Integer> intSeq = new Seq<Integer>(2004, null);
//  merge1(numSeq, intSeq); // Compile-time error
    merge2(numSeq, intSeq); // OK, T is Number
//  merge2(intSeq, numSeq); // Compile-time error
    merge3(intSeq, numSeq); // OK, T is Integer
//  merge3(numSeq, intSeq); // Compile-time error
```


Generic Methods and Wildcards

- Dependency between the type of the parameter `obj` and that of the element type of the collection `c` is not captured by the wildcard <?> in this nongeneric method:

```
static boolean add (Object obj, Collection<?> c) {
  return c.add(obj);   // Compile-time error:
                       // Cannot add to a collection of unknown type
}
```

- Dependency between the type of the parameter `obj` and that of the element type of the collection `c` is captured by the type parameter in this generic method.

```
static <T> boolean add (T obj, Collection<T> c) {
  return c.add(obj);   // OK
}
```

Mixing Raw Types and Generic Types

- For backward compatibility, use of the raw type of a generic type is allowed, i.e. a generic type can be used without its type parameters.
- Assignments from raw types to parameterized types generate a warning, as in (2).
- Using raw type references to call methods whose signature uses type parameters generates a warning, as in (5).
  - Note the `ClassCastException` in (6) at runtime because of the method call in (5).

```
import java.util.*;
public class Legacy {
  public static void main(String[] args) {
    List<String> strList = new ArrayList<String>(); // (0)
    List list = strList; // (1) Assignment to nongeneric reference is ok.
    strList = list;      // (2) warning: unchecked assignment
    strList.add("aha");  // (3) Method call typesafe.
//  strList.add(23);     // (4) Compile-time error
    list.add(23);        // (5) warning: [unchecked] unchecked call to add(E)
                         //     as a member of the raw type java.util.List
    System.out.println(strList.get(1).length()); // (6) ClassCastException
  }
}
```


OTHER IMPLICATIONS AND RESTRICTIONS REGARDING JAVA GENERICS

- A type parameter cannot be referenced in a static context.
- Runtime type check with `instanceof` and casting on generic types have no meaning.
- Arrays of generic types are not permitted.

```
Foo<?>[] ref1 = new Foo<?>[10];                      // OK
Foo<Integer>[] ref2 = new Foo<?>[10];                // Declaration not ok
Foo<? extends Number>[] ref3 = new Foo<Integer>[10]; // Not ok on both counts
```

- For overloading, it is not enough to use wildcards for formal parameter types.
- Generic classes can be extended:

```
class Parent<A, B> { /*...*/ }
class Child<X, Y, Z> extends Parent<Z, Comparable> { /*...*/ }
```

  - The subclass provides the actual type parameters for the superclass.
- For overriding, the return type of the method in the subclass can now be identical or a subtype of the return type of the method in the superclass.
- A generic class cannot be a subclass of `Throwable`.
- Type parameters are allowed in the `throws` clause, but not for the parameter of the `catch` block.
- Varargs with a parameterized type are not legal.