Rules for Generics

A *generic type* is a type (class or interface) with one or more type parameters. Some common examples are:

List<E>	a list (interface) containing elements of type E
ArrayList<E>	an ArrayList (class) containing elements of type E
Map<K,V>	a map with keys of type K and values of type V

The type parameter can be any valid Java variable name; by convention a single capital letter is generally used.

Instantiating a Generic Type

To create an instance of a generic type, you must supply a value for the type parameter(s). The value can the name of an Interface, Class, or Enum. It cannot be a primitive.

Valid: new ArrayList<Double>, new ArrayList<Comparable<Number>>

Invalid: new ArrayList<double> (can't use a primitive as type param)

Type Erasure

When you create an instance of a generic class, Java does not create a new kind of class, such as ArrayList<Double>. (C++ creates separate classes for each type used in the type parameter, such as ArrayList_Double, ArrayList_String, etc.) Java erases the type parameter and substitutes casts and type checks in your code to force compliance.

The result is that no extra classes are created and there is no run-time overhead for using generics.

Implementing a Generic Interface

When you implement an interface with a type parameter, Java requires that you substitute the actual type for the type parameter. For example:

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

If we implement Comparable<Foo>:

```
public class Foo implements Comparable<Foo> {
      // substitute "Foo" for "T"
      public int compareTo(Foo obj) { … }
}
```

Writing your own generic type

You can define classes with type parameters, like you did in the Stack class. The syntax is:

```
public class Stack<T> {
      private List<T> elements;
      public Stack( )
      public void push(T obj) . . .
      public T pop( ) …
```

as this example shows, you can use a generic type (T) as parameter, return type, or local variable type in your class.

However, you cannot create *instances* of a type parameter:

```
      public Stack( ) {
            T [] array = new T[3];  // error
            elements = new ArrayList<T>( );  // error
```

instead, create elements using Object or some known supertype and *cast* them:

```
      public Stack( ) {
      T [] array = (T[]) new Object[3];
      elements = (ArrayList<T>) new ArrayList<Object>( );
```

Static Methods and Generic Methods

A class's type parameter can only be used on instance members, not static members.

```
public class MyUtils<T> {
   private static T arg;    // error - static attribute
   public static void print(T a)    // error - static method
```

To use a type parameter in a static method, you must define a generic method. The syntax is:

```
            public static <T> return_type methodName( . . . )
```

Here's a static **sum( )** method to sum elements in a List of any numeric type:

```
public class MyUtils {
    public static <E extends Number> double sum(List<E> list) {
      int size = list.size();
      if (size == 0) return 0;
      return list.get(0).doubleValue() + sum(list.subList(1, size));
    }
```

Unfortunately, Java's Number is class is *lame*. It doesn't define any arithmetic operations like "add", "multiply", which would enable us to write more useful generic methods.

Bounds on Type Parameters

A plain type parameter such as List<T> accepts any class, interface, or enum as a value for T. You can restrict (bound) the possible value for the type parameter using keywords **super** and **extends**.

(1) **extends**: T can only be types that implements **Runnable**:

```
class TaskRunner<T extends Runnable> {
      private T task;
      public void doit( ) {
            task.run( );
```

In this example we can invoke **task.run( )** since task is type T and T is required be something that implements Runnable.

You can put **multiple bounds** on a type parameter by using **&**

```
class ObjectWriter <T extends Serializable & Cloneable> {
      // T must be a type that implements Serializable and Cloneable
```

If one of the bounds is a class then it must be specified first in the "extends" list:

```
class Foo { /* ordinary class */ }
class Bar<T extends Foo & Runnable>    // OK
class Bar<T extends Runnable & Foo>    // Error: "Foo" must be first
```

(2) **super** - require type parameter to be a superclass of a given type. This can only be used in conjunction with wildcards, discussed below.

Wildcard: ?

The ? is a wildcard type parameter. It means "any type", but can have bounds. It has a few uses.

1) the Set class has a method removeAll that removes all elements that are in the parameter collection:

```
      public boolean removeAll(Collection<?> coll)
```

this means "a collection of any type of element".

2) ? is often used with a bound. Consider a static sort method:

```
public static <E extends Comparable<E>> void sort(List<E> list)
```

it means "*E can be any type that implements Comparable<itself>*".

But what about a class that implements Comparable<some_superclass>?

For example, if BigDecimal implements Comparable<Number> than we would not be able to invoke the sort method using List<BigDecimal> as parameter. But all sort needs is for the type (E) to implement Comparable for some superclass of itself. Using wildcards we can write:

```
public static <E extends Comparable<? super E>> void sort(List<E> lst)
```

Many methods in the Collections and Arrays classes have parameters using bounded wildcards, for example, Collections.fill ("fill" a collection with a given element):

```
      public static <T> void fill(List<? super T> list, T obj)
```

Another example is Collections.binarySearch (find an element in a sorted collection):

```
     static <T> int binarySearch(List<? extends Comparable<? super T>> list, T key)
```

3) max combines **<? super T>** and **<? extends T>**

```
static <T extends Object & Comparable<? super T>> T
      max(Collection<? extends T> coll)
```

Invoking Generic Methods

To invoke a generic method you usually don't have to specify the type parameter. The compiler will figure it out from context. If you write:

```
List<Double> list = …
double result = MyUtils.sum( list );
```

Java will infer that **E** must be "**Double**".

However, you can explicitly specify the value of a generic method's type parameter using this ugly syntax:

```
double result = MyUtils.<Double>sum( list );
```

References

- *Object-Oriented Design and Patterns, 2E,* section 7.7
- Oracle *Java Tutorial*
- Langer's generics FAQ (info info about casting and subtypes involving type parameters) http://www.langer.camelot.de/GenericsFAQ/JavaGenericsFAQ.html