

The Class **class** itself is generic since Java 1.5.

```
public final class Class<T> extends Object
    implements Serializable, GenericDeclaration,
Type, AnnotatedElement
{
    ...
}
```

The **T** type here represents the type that is handed to the Class object. The **T** type will be substituted with the class being loaded.

Class<T>

Since Java 1.5, the class `java.lang.Class` is generic. It is an interesting example of using genericness for something other than a container class.

For example, the type of `String.class` is `Class<String>`, and the type of `Serializable.class` is `Class<Serializable>`. This can be used to improve the type safety of your **reflection code**.

In particular, since the `newInstance()` method in `Class` now returns a `T`, you can get more precise types when creating objects reflectively.

Now we can use the `newInstance()` method to return a new object with exact type, without casting.

```
Customer cust = Utility.createAnyObject (Customer.class); // - No
casting
...
public static <T> T createAnyObject (Class<T> cls)
{
    T ret = null;
    try
    {
        ret = cls.newInstance();
    }
    catch (Exception e)
    {
        // --- Exception Handling
    }
    return ret;
}
```

And the above code without Generics:

```
Customer cust = (Customer) Utility.createAnyObject (Customer.class); //
- Casting is needed
...
public static Object createAnyObject (Class cls)
{
    Object ret = null;
    try
    {
        ret = cls.newInstance();
    }
```

```
    }  
    catch (Exception e)  
    {  
        // --- Exception Handling  
    }  
    return ret;  
}
```

Get exact type when getting JavaBean property, using reflection

See the following code where the method will return the exact type of the Java Bean property, based on how it will be called.

```
// --- Using reflection, get a Java Bean property by its name ---  
public static <T> T getProperty(Object bean, String propertyName)  
{  
    if (bean == null ||  
        propertyName == null ||  
        propertyName.length() == 0)  
    {  
        return null;  
    }  
    // --- Based on the property name build the getter method name ---  
    String methodName = "get" +  
        propertyName.substring(0,1).toUpperCase() +  
        propertyName.substring(1);  
    T property = null;  
    try  
    {  
        java.lang.Class c = bean.getClass();  
        java.lang.reflect.Method m = c.getMethod(methodName, null);  
        property = (T) m.invoke(bean, null);  
    }  
    catch (Exception e)  
    {  
        // --- Handle exception --  
    }  
    return property;  
}
```

Variable Argument

Using Generics, it is very easy to define a method with a variable number of arguments. Before generics, this was not possible in Java—if a likewise feature was needed, this was mostly done by passing an array. The only requirement for using a variable number of arguments using Generics is that the arguments in the list must have the same type.

The following code illustrates a method that can be called with a variable number arguments:

```
/**
 * Method using variable-length argument list
 * @param <T>
 * @param args
 */
public static <T> List<T> makeAList(T... args)
{
    List<T> argList = new ArrayList<T>();
    for (int i = 0; i < args.length; i++)
    {
        argList.add(args[i]);
    }
    return argList;
}
```

The above method can be called with a variable number of arguments, for example:

```
List<String> list1 = makeAList("One", "Two", "Three");
List<String> list2 = makeAList("One", "Two", "Three", "Four");
```

In the above example calls, the arguments must be of type String. If we write `<? extends Object>` instead of `T`, then we can pass *any* kind of objects, regardless of their type:

```
List<? extends Object> list3 = makeAList("One", 10, new
StringBuffer(), new LinkedList());
```

Note: the number 10 in the above code will be converted (autoboxed) to Integer.

See also:

```
java.util.Arrays.asList(T... a)
```

Wildcard Types

As we have seen above, generics give the impression that a new container type is created with each different type parameter. We have also seen that in addition to the normal type checking, the type parameter has to match as well when we assign generics variables.

In some cases this is too restrictive. What if we would like to relax this additional checking? What if we would like to define a collection variable that can hold any generic collection, regardless of the parameter type it holds?

Wildcard

The wildcard type is represented by the character `<?>`, and pronounced **Unknown**, or **Any-Type**. This **Unknown** type matches anything, if it is used only by itself. Any-Type can be express also by `<? extends Object>`. Any-Type includes Interfaces, not only Classes.

So now we can define a collection whose element type matches anything. See below: