Handling Java Generic Types with Reflection

When declaring a class or an interface as being parameterizable, or when a subclass extends that parameterizable class, we come into cases where we need to access the information of those generic parameters, this is where **Java Reflection** comes in handy.
This tutorial will introduce `ReflectionUtil`, a utility class that uses `java.lang.reflect` standard library. It provides easy access to the standard reflect methods that are needed usually when dealing with generic object types (`java.reflect.Type`).

This article assumes you have some good knowledge of reflection API and the common uses of it. If not, you might find Oracle's Reflection API Tutorial a helpful start.
`ReflectionUtil` covers the next functionalities:

1. Get `Class` object by `Type`
2. Create instance by `Type`
3. Get `Types` of a parameterized generic object

1. Get `Class` object by Type

```
1  private static final String T
2
3  public static String getClass
4      if (type==null) {
5          return "";
6      }
7      String className = type.t
8      if (className.startsWith(
9          className = className
10     }
11     return className;
12 }
13
14 public static Class<?> getCla
15             throws ClassNotFo
16     String className = getCla
17     if (className==null || cl
18         return null;
19     }
20     return Class.forName(clas
21 }
```

Method `ReflectionUtil#getClass(Type)` is used to get `java.lang.Class` object from `java.lang.reflect.Type`. This method takes advantage of the `toString()` value from a `Type` which

`toString()` value from a `Type` which gives the fully qualified name of a class as "`class some.package.Foo`". `ReflectionUtil#getClassName(Type)` substrings this last value removing the prefix "class " (with the space) to make it eligible for `ReflectionUtil#getClass(Type)` that in its turn uses `Class.forName(String)` to load the desired class properly.

2. Create instance by Type

```
1  public static Object newInstan
2          throws ClassNotFoundEx
3      Class<?> clazz = getClass(
4      if (clazz==null) {
5          return null;
6      }
7      return clazz.newInstance()
8  }
```

Method `ReflectionUtil#newInstance(Type type)` creates a newly allocated instance of the class represented by the invoked `Type` object. The given `Type` should not represents an abstract class, an interface, an array class, a primitive type, or void, otherwise an `InstantiationException` is thrown.

3. Get Types of a parameterized generic object

Let's say we have the next objects:

Let's say we have the next objects:

```
1  public abstract class Foo<T> {
2      //content
3  }
4
5  public class FooChild extends
6      //content
7  }
```

What if we needed `Class<T>` object inside `Foo`?

This can be solved in either two ways:

- The common way of forcing `FooChild` to pass its own `Class` as next:

```
1   public abstract class Foo<T>
2       private Class<T> tClass;
3
4       public Foo(Class<T> tClas
5           this.tClass = tClass;
6       }
7       //content
8   }
9
10  public class FooChild extends
11      public FooChild() {
12          super(FooChild.class)
13      }
14      //content
15  }
```

- Or using **reflection**:

```
1  public static Type[] getParame
2      Type superclassType = obje
3      if (!ParameterizedType.cla
4          return null;
5      }
6      return ((ParameterizedType
7  }
```

Method `ReflectionUtil#getParameterizedTypes(Object)` returns an array of `Type[]` objects representing the actual type arguments to this object, which is the `Type` of `T` at

representing the actual type arguments to this object, which is the `Type` of `T` at runtime as in our example.
So in order for `Foo` to get `Class<T>`, it will be using `ReflectionUtil#getParameterizedTypes` accompanied with `ReflectionUtil#getClass` as next:

```
...
Type[] parameterizedTypes = Re
Class<T> clazz = (Class<T>)Ref
...
```

You should also note this from the `java.lang.reflect.ParameterizedType#getActualTypeArgument` documentation:

> in some cases, the returned array can be empty. This can occur
> if this type represents a non-parameterized type nested within
> a parameterized type.

---

4. Check if a `Class` has a default constructor

```
public static boolean hasDefau
    Class<?>[] empty = {};
    try {
        clazz.getConstructor(e
    } catch (NoSuchMethodExcep
        return false;
    }
    return true;
}
```

```
9 }
```

Method `ReflectionUtil#hasDefaultConstructor` checks whether a `java.lang.reflect.Constructor` object with no parameter types is specified by the invoked `Class` object or not.

5. Get `Class` object representing the type of a `Field` in a class

```
1  public static Class<?> getFie
2      if (clazz==null || name==
3          return null;
4      }
5      name = name.toLowerCase()
6      Class<?> propertyClass =
7      for (Field field : clazz.
8          field.setAccessible(t
9          if (field.getName().e
10             propertyClass = f
11             break;
12         }
13     }
14     return propertyClass;
15 }
```

In some cases you'll be needing to get the `Class` object type of a declared field inside a given class knowing only the `Class` object containing the field, and the field's `String` name.

Method `ReflectionUtil#getFieldClass(Class<?>, String)` loops through `Class#getDeclaredFields()` comparing each `java.lang.reflect.Field#getName()`

comparing each `java.lang.reflect.Field#getName()` with the invoked name, when a match is found, `Field#getType()` is returned with the `Class` object needed which represents the type of our field.

6. Get `Class` object representing the return type of a `Method` in a class

```
1  public static Class<?> getMet
2      if (clazz==null || name==
3          return null;
4      }
5
6      name = name.toLowerCase()
7      Class<?> returnType = nul
8
9      for (Method method : claz
10         if (method.getName().
11             returnType = meth
12             break;
13         }
14     }
15
16     return returnType;
17 }
```

When you need to know the return type `Class` object of a declared method inside a given class knowing only the `Class` object containing the method, and the method's `String` name.
Method `ReflectionUtil#getMethodReturnType(Class<?>, String)` loops through `Class#getDeclaredMethods()` comparing each `java.lang.reflect.Method#getName()` with the invoked name, when a match is found, `Method#getReturnType()` is

with the invoked name, when a match is found, `Method#getReturnType()` is returned with the `Class` object needed which represents the return type of our method.

7. Get `Enum` constant of an enum `Class` by `String` identifier

```
1  @SuppressWarnings({ "unchecked
2  public static Object getEnumCo
3      if (clazz==null || name==n
4          return null;
5      }
6      return Enum.valueOf((Class
7  }
```

Method `ReflectionUtil#getEnumConstant(Class<?>, String)` extracts the enum constant of the specified enum class with the specified name. The name must match exactly an identifier used to declare an enum constant in the given class.
This is useful when your representing enum `Class<?>` is of a generic type.

8. Download `ReflectionUtil` class

Get the full and documented `ReflectionUtil.java` and comment out to tell me of any improvements or add ups to this utility.

Copyrights Notice: You can freely use class ReflectionUtil.java or any code of

Copyrights Notice: You can freely use class ReflectionUtil.java or any code of this tutorial in your application or modify it to your needs, just keeping my name and the link would be fair.