Reflection

TypeToken

Due to type erasure, you can't pass around generic `Class` objects at runtime – you might be able to cast them and pretend they're generic, but they really aren't.

For example:

```
ArrayList<String> stringList = Lists.newArrayList();
ArrayList<Integer> intList = Lists.newArrayList();
System.out.println(stringList.getClass().isAssignableFrom(intList.getClass()));
// returns true, even though ArrayList<String> is not assignable from ArrayList<Integer>
```

Guava provides `TypeToken`, which uses reflection-based tricks to allow you to manipulate and query generic types, even at runtime. **Think of a `TypeToken` as a way of creating, manipulating, and querying `Type` (and, implicitly `Class`) objects in a way that respects generics.**

Note to Guice users: `TypeToken` is similar to Guice's `TypeLiteral` class, but with one important difference: it supports non-reified types such as `T`, `List<T>` or even `List<? extends Number>`; while `TypeLiteral` does not. `TypeToken` is also serializable and offers numerous additional utility methods.

Background: Type Erasure and Reflection Java doesn't retain generic type information for *objects* at runtime. If you have an `ArrayList<String>` object at runtime, you cannot determine that it had the generic type `ArrayList<String>` – and you can, with unsafe raw types, cast it to `ArrayList<Object>`.

However, reflection allows you to detect the generic types of methods and classes. If you implement a method that returns a `List<String>`, and you use reflection to obtain the return type of that method, you get back a `ParameterizedType` representing `List<String>`.

The `TypeToken` class uses this workaround to allow the manipulation of generic types with a minimum of syntactic overhead.

Introduction Obtaining a `TypeToken` for a basic, raw class is as simple as

```
TypeToken<String> stringTok = TypeToken.of(String.class);
TypeToken<Integer> intTok = TypeToken.of(Integer.class);
```

To obtain a `TypeToken` for a type with generics – when you know the generic type arguments at compile time – you use an empty anonymous inner class:

```
TypeToken<List<String>> stringListTok = new TypeToken<List<String>>() {};
```

Or if you want to deliberately refer to a wildcard type:

```java
TypeToken<Map<?, ?>> wildMapTok = new TypeToken<Map<?, ?>>() {};
```

TypeToken provides a way to dynamically resolve generic type arguments, like this:

```java
static <K, V> TypeToken<Map<K, V>> mapToken(TypeToken<K> keyToken, TypeToken<V> valueToken)
  return new TypeToken<Map<K, V>>() {}
    .where(new TypeParameter<K>() {}, keyToken)
    .where(new TypeParameter<V>() {}, valueToken);
}
...
TypeToken<Map<String, BigInteger>> mapToken = mapToken(
   TypeToken.of(String.class),
   TypeToken.of(BigInteger.class));
TypeToken<Map<Integer, Queue<String>>> complexToken = mapToken(
   TypeToken.of(Integer.class),
   new TypeToken<Queue<String>>() {});
```

Note that if `mapToken` just returned `new TypeToken<Map<K, V>>()`, it could not actually reify the types assigned to `K` and `V`, so for example

```java
class Util {
  static <K, V> TypeToken<Map<K, V>> incorrectMapToken() {
    return new TypeToken<Map<K, V>>() {};
  }
}

System.out.println(Util.<String, BigInteger>incorrectMapToken());
// just prints out "java.util.Map<K, V>"
```

Alternately, you can capture a generic type with a (usually anonymous) subclass and resolve it against a context class that knows what the type parameters are.

```java
abstract class IKnowMyType<T> {
  TypeToken<T> type = new TypeToken<T>(getClass()) {};
}
...
new IKnowMyType<String>() {}.type; // returns a correct TypeToken<String>
```

With this technique, you can, for example, get classes that know their element types.

Queries

`TypeToken` supports many of the queries supported by `Class`, but with generic constraints properly taken into account.

Supported query operations include:

Method	Description
getType()	Returns the wrapped java.lang.reflect.Type.
getRawType()	Returns the most-known runtime class.
getSubtype(Class<?>)	Returns some subtype of this that has the specified raw class. For example, if this is Iterable<String> and the argument is List.class, the result will be List<String>.
getSupertype(Class<?>)	Generifies the specified raw class to be a supertype of this type. For example, if this is Set<String> and the argument is Iterable.class, the result will be Iterable<String>.
isSupertypeOf(type)	Returns true if this type is a supertype of the given type. "Supertype" is defined according to the rules for type arguments introduced with Java generics.
getTypes()	Returns the set of all classes and interfaces that this type is or is a subtype of. The returned Set also provides methods classes() and interfaces() to let you view only the superclasses and superinterfaces.
isArray()	Checks if this type is known to be an array, such as int[] or even <? extends A[]>.

Method	Description
`getComponentType()`	Returns the array component type.

resolveType `resolveType` is a powerful but complex query operation that can be used to "substitute" type arguments from the context token. For example,

```
TypeToken<Function<Integer, String>> funToken = new TypeToken<Function<Integer, String>>()

TypeToken<?> funResultToken = funToken.resolveType(Function.class.getTypeParameters()[1]));
  // returns a TypeToken<String>
```

`TypeToken` unifies the `TypeVariable`s provided by Java with the values of those type variables from the "context" token. This can be used to generically deduce the return types of methods on a type:

```
TypeToken<Map<String, Integer>> mapToken = new TypeToken<Map<String, Integer>>() {};
TypeToken<?> entrySetToken = mapToken.resolveType(Map.class.getMethod("entrySet").getGeneric
  // returns a TypeToken<Set<Map.Entry<String, Integer>>>
```

Invokable

Guava's `Invokable` is a fluent wrapper of `java.lang.reflect.Method` and `java.lang.reflect.Constructor`. It simplifies common reflective code using either. Some usage examples follow:

Is the method public? JDK:

```
Modifier.isPublic(method.getModifiers())
```

`Invokable`:

```
invokable.isPublic()
```

Is the method package private? JDK:

```
!(Modifier.isPrivate(method.getModifiers()) || Modifier.isPublic(method.getModifiers()))
```

`Invokable`:

```
invokable.isPackagePrivate()
```

Can the method be overridden by subclasses? JDK:

```
!(Modifier.isFinal(method.getModifiers())
    || Modifiers.isPrivate(method.getModifiers())
    || Modifiers.isStatic(method.getModifiers())
    || Modifiers.isFinal(method.getDeclaringClass().getModifiers()))
```

Invokable:

```
invokable.isOverridable()
```

Is the first parameter of the method annotated with @Nullable? JDK:

```
for (Annotation annotation : method.getParameterAnnotations()[0]) {
  if (annotation instanceof Nullable) {
    return true;
  }
}
return false;
```

Invokable:

```
invokable.getParameters().get(0).isAnnotationPresent(Nullable.class)
```

How to share the same code for both constructors and factory methods? Are you tempted to repeat yourself because your reflective code needs to work for both constructors and factory methods in the same way?

`Invokable` offers an abstraction. The following code works with either Method or Constructor:

```
invokable.isPublic();
invokable.getParameters();
invokable.invoke(object, args);
```

What's the return type of List.get(int) for List<String>? `Invokable` provides type resolution out of the box:

```
Invokable<List<String>, ?> invokable = new TypeToken<List<String>>() {}.method(getMethod);
invokable.getReturnType(); // String.class
```

Dynamic Proxies

newProxy() Utility method `Reflection.newProxy(Class, InvocationHandler)` is a more type safe and convenient API to create Java dynamic proxies when only a single interface type is to be proxied.

JDK:

```
Foo foo = (Foo) Proxy.newProxyInstance(
    Foo.class.getClassLoader(),
    new Class<?>[] {Foo.class},
    invocationHandler);
```

Guava:

```
Foo foo = Reflection.newProxy(Foo.class, invocationHandler);
```

AbstractInvocationHandler Sometimes you may want your dynamic proxy to support equals(), hashCode() and toString() in the intuitive way, that is: * A proxy instance is equal to another proxy instance if they are for the same interface types and have equal invocation handlers. * A proxy's toString() delegates to the invocation handler's toString() for easier customization.

`AbstractInvocationHandler` implements this logic.

In addition, `AbstractInvocationHandler` ensures that the argument array passed to `handleInvocation(Object, Method, Object[])` is never null, thus less chance of `NullPointerException`.

ClassPath

Strictly speaking, Java has no platform-independent way to browse through classes or class path resources. It is however sometimes desirable to be able to go through all classes under a certain package or project, for example, to check that certain project convention or constraint is being followed.

`ClassPath` is a utility that offers best-effort class path scanning. Usage is simple:

```
ClassPath classpath = ClassPath.from(classloader); // scans the class path used by classloa
for (ClassPath.ClassInfo classInfo : classpath.getTopLevelClasses("com.mycomp.mypackage")) {
  ...
}
```

In the above example, `ClassInfo` is a handle to the class to be loaded. It allows programmers to check the class name or package name and only load the class until necessary.

It's worth noting that `ClassPath` is a best-effort utility. It only scans classes in jar files or under a file system directory. Neither can it scan classes managed by custom class loaders that aren't URLClassLoader. So **don't use it for mission critical production tasks**.

Class Loading

The utility method `Reflection.initialize(Class...)` ensures that the specified classes are initialized – for example, any static initialization is performed.

The use of this method is a code smell, because static state hurts system maintainability and testability. In cases when you have no choice while interoperating with a legacy framework, this method helps to keep the code less ugly.