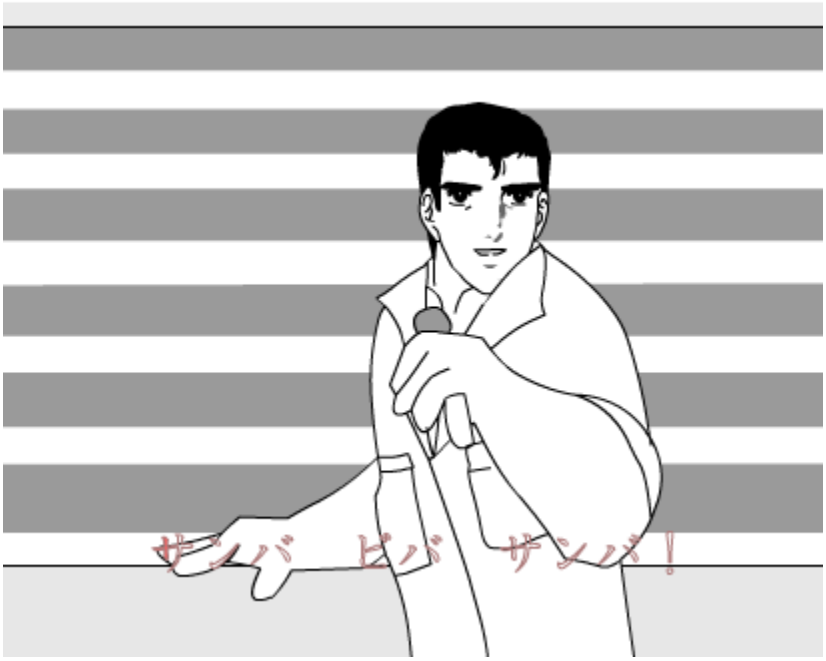
수퍼 타입 토큰 뽀개기 (어랏.. 잘 안뽀개지네...)

혹시 수퍼 타입 토큰이라는 용어를 들어보셨나요?



저는 수퍼 타입 토큰을 사용해봤지만 정확한 동작원리를 알지는 못했었습니다. ㅎㅎ

그래서 답답한 마음이 있었고 수퍼 타입 토큰이 무엇이고 어떻게 활용될 수 있는지 처음부터 쭉 조사하고 정리해보는 시간을 가져봤습니다.

의식의 흐름에 맡겨 정리해보고 싶어 궁금했던 의문점들을 목차로 만들었습니다

의문점에 대한 답을 찾아가는 형식으로 쭉 정리해보겠습니다.

목차

험난한 수퍼 타입 토큰 뽀개기 여정을 시작해 보겠습니다~!

본론

1. JAVA 제네릭(Generic)은 뭐죠?

- 클래스에서 사용할 타입을 외부에서 정하는 것을 의미합니다.
- 선언시 클래스 또는 인터페이스에 '< >' 기호를 쓰고 그안에 타입을 넣으면 됩니다.
- 제네릭을 사용함으로써 Object형을 강제 형변환할 필요가 줄어들었습니다.

사용 예시

```
List<String> stringList = new ArrayList<>();
```

- 제네릭 사용 유무에 따른 테스트를 진행해보겠습니다.

제네릭을 사용한 list

```
public class GenericListSample {
    public List<String> list = new ArrayList<>();

    public void addString(String str) {
        list.add(str);
    }

    public String getString(int index) {
        return list.get(index);
    }
}
```

제네릭을 사용 안한 list

```
public class NonGenericListSample {
    public List list = new ArrayList();

    public void addString(Object obj) {
        list.add(obj);
    }

    public String getString(int index) {
        return (String) list.get(index);
    }
}
```

- 위에서 정의한 리스트 2개를 테스트 해보겠습니다.

테스트 코드

```
class GenericListSampleTest {
    private GenericListSample list = new GenericListSample();

    @Test
    public void 리스트에_String_삽입하고_가져오기_테스트() {
        String mockStr = "문자열1";
        list.addString(mockStr);
        String str = list.getString(0);
        assertEquals(str, mockStr);
    }

    @Test
    public void 리스트에_Integer_삽입하고_가져오기_테스트() {
        Integer mockInt = 1;
//          list.addString(mockInt);   //컴파일 타임에 오류 감지
        String str = list.getString(0);
        assertEquals(str, mockInt);
    }
}



class NonGenericListSampleTest {
    private NonGenericListSample list = new NonGenericListSample();

    @Test
    public void 리스트에_String_삽입하고_가져오기_테스트() {
        String mockStr = "문자열1";
        list.addString(mockStr);
        String str = list.getString(0);
        assertEquals(str, mockStr);
    }

    @Test
    public void 리스트에_Integer_삽입하고_가져올때_ClassCastException_발생_테스트() {
        Integer mockInt = 1;
        list.addString(mockInt);   //컴파일 타임에 오류 감지 못함
        //아래 처럼 런타임때 ClassCastException이란 예외 발생!
        Assertions.assertThrows(ClassCastException.class, () -> {
            String obj = list.getString(0);
        });
    }
}
```

2. JAVA 제네릭은 완전할까요?

- 자바에 제네릭이 추가되면서 컴파일 시기때 타입 안정성을 얻을 수 있었습니다.
- 이 덕분에 IDE에서도 타입이 맞지 않으면 미리 경고 해줄 수 있습니다.
- 무시하고 컴파일을 한다면 당연스럽게 컴파일에러가 발생합니다.
- 다만 혹자는 자바의 제네릭이 반쪽짜리 제네릭이라고 하기도 합니다. 그 이유는 자바의 런타임 타입 안정성과 관련이 있습니다

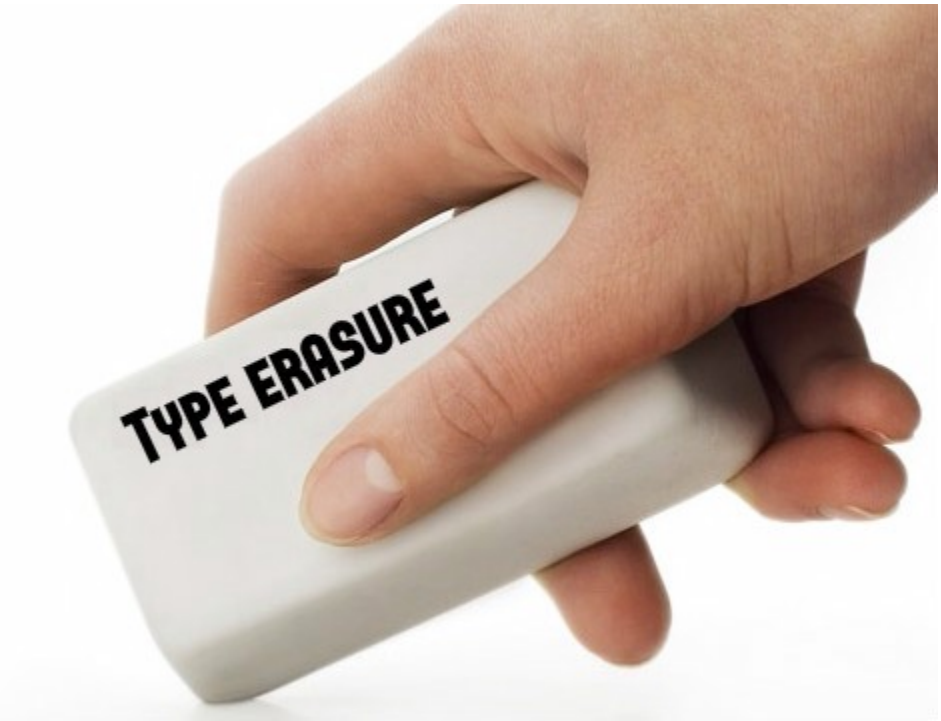
2_1. JAVA와 C#의 제네릭 비교

Parameters	Java	C#
Creation	Designed by Sun Microsystems.	Designed as part of Microsoft's .NET initiative.
Ecosystem	Has a huge opensource ecosystem.	Used to develop software for Microsoft platforms.
Support for generics	It is implemented using erasures and casts added upon compilation into bytecode.	Integrated into the CLI and allows type information to be available at runtime

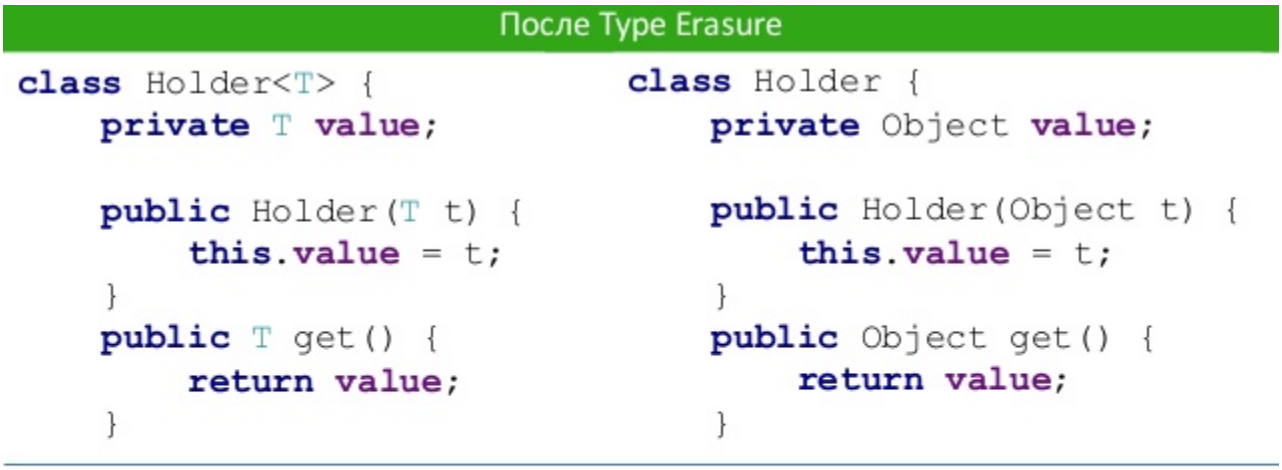
출처: https://www.guru99.com/java-vs-c-sharp-key-difference.html

- JAVA의 제네릭은 런타임 타입 안정성이 없습니다. 컴파일 과정에서 Type Erasure(타입 소거자) 통해 타입 파라미터를 전부 지웠기 때문입니다.

2_2. Type Erasure



- 타입 소거자는 아래 그림과 같이 컴파일 과정에서 **타입**을 **소거**하고 **Object**로 만듭니다.



2_3. 그렇다면 JAVA는 왜 Type Erasure를 사용했나요?

제네릭을 도입할 당시 JAVA와 C#은 보급률에 큰 차이가 있었습니다. JAVA로 구현된 현업 프로젝트가 월등히 많았던 것입니다. JAVA는 하위 호완성을 지키는 언어로 많은 사랑을 받았고 이를 꼭 지키고자 했습니다. 때문에 자바 진영은 컴파일 과정에서 Type Erasure 과정을 통해 타입 파라미터를 전부 지워주고 제네릭이 없던 하위 버전과 동일한 형태로 class 파일을 생성합니다. 반면 C#은 당시 보급률이 현저히 낮았기 때문에 하위 호완성을 포기하고 컴파일, 런타임 시기에 모두 완전한 제네릭을 적용했습니다. 다만 이를 보완하기 위해 C#에서도 기계적 마이그레이션을 할 수 있도록 툴을 제공했다고 합니다.

2_4. 이 문제를 어떻게 해결할 수 있을까요?

자바의 Type Erasure로 인해 제네릭 타입이 런타임때 없어지는 문제로 런타임 때 타입 안정성을 확보할 수 없는 문제가 있습니다. 자바에서 이 문제를 수퍼 타입 토큰을 이용해 해결할 수 있습니다. 이를 위해 타입 토큰부터 슈퍼 타입 토큰에 대해 전반적으로 알아보겠습니다.

3. 타입 토큰(Type Token) 이란 무엇일까요?

3_1. 타입 토큰을 정의한다면?

자바언어 개발자였던 Neal Gafter는 JAVA JDK5에 generics를 추가할 때 java.lang.Class 가 generic type이 되도록 변경했다고 합니다. 예를들어, String.class의 Type이 Class<String> 되도록 한 것이라고 합니다. 또한 이를 명칭하기 위해 Gilad Bracha라는 분이 타입 토큰이라는 용어를 만들어 줬다고 합니다. 토큰의 전산적 의미를 고려한다면 타입 토큰은 이런 뜻이라고 조심스레 유추해봅니다 ㅎㅎ

" "

3_2. 클래스 리터럴과 타입 토큰의 의미는?

- 클래스 리터럴(Class Literal)은 String.class, Integer.class 등을 말합니다.
- String.class의 타입은 Class<String>, Integer.class의 타입은 Class<Integer>입니다.
- 클래스 리터럴은 타입 토큰으로서 사용됩니다. Class.class 와 Class<String> 가 동일하다고 생각하면 좀 더 이해가 될 것 같습니다.
- 아래는 클래스 리터럴과 타입 토큰의 관계를 보여주는 예시입니다.

클래스 리터럴과 타입 토큰의 관계 예시

```
// 아래와 같은 메서드는 타입 토큰을 인자로 받는 메서드입니다.
void myMethod(Class<?> clazz) {
  ...
}

// String.class라는 클래스 리터럴을 타입 토큰 파라미터로 myMethod에 전달합니다.
myMethod(String.class);
```

3_3. 타입 토큰은 어디에 쓰이나요?

- 주로 타입 토큰은 타입 안전성이 필요한 곳에 사용됩니다.

타입 토큰 사용 예시

```
//ObjectMapper의 readValue 메서드 파라미터로 String 과 클래스 리터럴을 전달합니다.
ProductDto productDto = objectMapper.readValue(jsonString, ProductDto.class);

//전달된 클래스 리터럴인 ProductDto.class를 타입토큰인 Class<T> valueType로 받고 있습니다.
public <T> T readValue(String content, Class<T> valueType) {
...
}
```

3_4. 타입 토큰의 한계점은?

- 한계점을 알아보기 위해 THC(Typesafe Heterogenous Container) pattern을 사용한 예제를 만들어 보겠습니다.
- 위 패턴이 적용된 SimpleTypeSafeMap을 만들어 보겠습니다.

THC 패턴이 적용된 SimpleTypeSafeMap

```
public class SimpleTypeSafeMap {
    private Map<Class<?>, Object> map = new HashMap<>();

    public <T> void put(Class<T> k, T v) {
        map.put(k, v);
    }

    public <T> T get(Class<T> clazz) {
        return clazz.cast(map.get(clazz));
    }
}
```

- 타입 토큰을 이용해서 별도의 캐스팅 없이도 타입 안전성을 확보할 수 있게 됐습니다.

simpleTypeSafeMap put, get test

```
@Test
public void type_token을_이용한_put_get_테스트() {
        simpleTypeSafeMap.put(String.class, "11st_문자열");
        simpleTypeSafeMap.put(Integer.class, 11);

        //타입 토큰을 이용해서 별도의 캐스팅 없이도 타입 안전성이 확보가능합니다.
        String v1 = simpleTypeSafeMap.get(String.class);
        Integer v2 = simpleTypeSafeMap.get(Integer.class);

        assertTrue(v1 instanceof String);
        assertTrue(v2 instanceof Integer);
}
```

- 하지만 List<String>.class와 같은 클래스 리터럴은 존재하지 않습니다. 타입 이레이저에 의해 이런 형식의 타입 토큰을 사용할 수 없다는 한계를 가지고 있습니다.

type token 한계 확인용 test

```
@Test
public void type_token_한계() {
        simpleTypeSafeMap.put(List.class, Arrays.asList(1,2,3));
        simpleTypeSafeMap.put(List.class, Arrays.asList("1", "2", "3"));

        List<Integer> v1 = (List<Integer>)simpleTypeSafeMap.get(List.class);
        List<String> v2 = (List<String>)simpleTypeSafeMap.get(List.class);

        assertFalse(v1.get(0) instanceof Integer);
        assertTrue(v2.get(0) instanceof String);
}
```

- 이런 한계를 극복할 수 있는 해결책이 바로 **수퍼 타입 토큰** 입니다.

4. 수퍼 타입 토큰(Super Type Token)으로 한계를 극복해 보아요!

4_1. 수퍼 타입 토큰이란?

- 타입 토큰계의 슈퍼맨?

type erasure 의 공격에도 꿈쩍 없는 슈느님!

- 수퍼 타입 토큰은 Neal Gafter라는 사람이 처음 고안한 방법으로 알려져 있습니다. 수퍼급의 타입 토큰이 아니라, 수퍼 타입을 토큰으로 사용한다는 의미입니다.
- 수퍼 타입 토큰은 상속과 Reflection을 기발하게 조합해서 List<String>.class 같이, 원래는 사용할 수 없었던 클래스 리터럴을 타입 토큰으로 사용하는 것과 같은 효과를 낼 수 있습니다.
- 앞에서 클래스 리터럴을 설명할 때, String.class의 타입이 Class<String>이라고 했었습니다. Class<String>이라는 타입 정보를 String.class라는 클래스 리터럴로 구할 수 있었던 덕분에 타입 안전성을 확보할 수 있었습니다.
- List<String>.class도 타입을 구할 수만 있다면 타입 안전성을 확보할 수 있다는 것은 마찬가지 입니다. 다만 Class<List<String>>라는 타입은 List<String>.class 같은 클래스 리터럴로 쉽게 구할 수 없다는 점이 다릅니다. 하지만 어떻게든 Class<List<String>>라는 타입을 구할 수 있다면, 타입 안전성을 확보할 수 있습니다.

4_2. List 내 타입과 같은 중첩된 타입은 어떻게 구할 수 있나요?

- Class.getGenericSuperclass()와 ParameterizedType.getActualTypeArguments()를 사용하면 됩니다!
- Class.getGenericSuperclass() API 문서를 보면 아래 정보를 알 수 있습니다.

getGenericSuperclass 메서드

```
// Class.class 내 getGenericSuperclass 메서드
/*
* 상위의 수퍼 클래스의 타입을 반환하며,
* 상위의 수퍼 클래스가 ParameterizedType이면, 실제 타입 파라미터들을 반영한 타입을 반환해야 한다.
* ParameterizedType에 대해서는 별도 문서를 참고하라.
*/
 public Type getGenericSuperclass() {
      ClassRepository info = this.getGenericInfo();
      if (info == null) {
          return this.getSuperclass();
      } else {
          return this.isInterface() ? null : info.getSuperclass();
      }
  }
```

- getGenericSuperclass()를 이용해 출력해보기

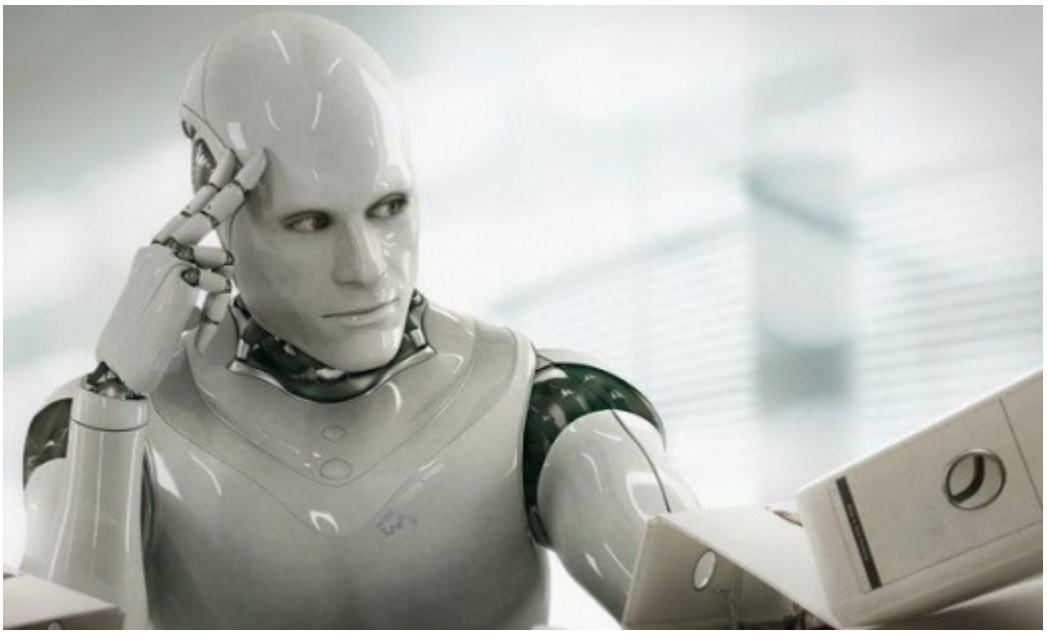
getGenericSuperclass를 사용해 출력해보는 테스트 코드

```
@Test
public void getGenericSuperclass_반환형인_Type_출력() {
        class Super<T> {}
        class MyClass extends Super<List<String>> {}  // 수퍼 클래스에 사용되는 파라미터 타입을 이용한다. 그래서 수퍼 타입 토큰.

        MyClass myClass = new MyClass();

        // 파라미터 타입 정보가 포함된 수퍼 클래스의 타입 정보를 구한다.
        Type typeOfGenericSuperclass = myClass.getClass().getGenericSuperclass();

        // ~~~$1Super<java.util.List<java.lang.String>> 출력됨
        System.out.println(typeOfGenericSuperclass);
}
```

- ParameterizedType.getActualTypeArguments()
- 위에 getGenericSuperclass()의 docs 설명을 보면, 수퍼 클래스가 ParameterizedType이면 타입 파라미터를 포함한 정보를 반환해야 한다고 했으며, ParameterizedType은 별도의 문서를 확인하라고 했습니다.쉽게 말하자면 ParameterizedType은 제네릭 타입을 가지고 있는 Type입니다.
- ParameterizedType의 API 문서 를 보면 Type[] getActualTypeArgumensts()라는 메서드가 있음을 확인할 수 있습니다. 위에서 말씀드린 것 터럼 제네릭 타입이 있는 타입이기 때문에 제네릭의 실제 타입을 가져올 수 있습니다.
- getActualTypeArguments()를 이용해 출력해보기

getActualTypeArguments를 사용해 출력해보는 테스트 코드

```
@Test
public void getActualTypeArguments_Type_출력() {
        class Super<T> {}
        class MyClass extends Super<List<String>> {}  // 수퍼 클래스에 사용되는 파라미터 타입을 이용한다. 그래서 수퍼 타입 토큰.

        MyClass myClass = new MyClass();

        // 파라미터 타입 정보가 포함된 수퍼 클래스의 타입 정보를 구한다.
        Type typeOfGenericSuperclass = myClass.getClass().getGenericSuperclass();

        // ~~~$1Super<java.util.List<java.lang.String>> 출력됨
        System.out.println(typeOfGenericSuperclass);

        // 수퍼 클래스가 ParameterizedType 이므로 ParameterizedType으로 캐스팅 가능
        // ParameterizedType의 getActualTypeArguments()으로 실제 타입 파라미터의 정보를 구함
        Type actualType = ((ParameterizedType) typeOfGenericSuperclass).getActualTypeArguments()[0];

        // 원했던 정보였던 java.util.List<java.lang.String>가 출력됨
        System.out.println(actualType);
}
```



드뎌! 제네릭 정보를 알 수 있게 됐습니다. 이제는 수퍼 타입 토큰을 이용해 더욱 안전한 TypeSafeMap을 만들 수 있을 것 같습니다!

4_3. super type token을 이용한 TypeSafeMap 만들어 보아요!

- 이전에 만들었던 SimpleTypeSafeMap은 중첩된 타입토큰을 얻지 못하는 한계가 있었습니다. super type token을 이용해서 한계를 극복한 Map을 만들어 보겠습니다.
- TypeSafe한 Map을 만들기 위해 Type 정보를 저장할 TypeReference를 만듭니다. TypeSafeMap을 만들어 TypeReference가 가지고 있는 Type을 이용합니다.
- TypeReference: 수퍼 타입 토큰을 저장하기 위한 추상클래스를 만들었습니다.

TypeReference

```
import java.lang.reflect.ParameterizedType;
import java.lang.reflect.Type;

public abstract class TypeReference<T> {
    private final Type type;

    protected TypeReference() {
        Type superClassType = getClass().getGenericSuperclass();
        if (superClassType instanceof ParameterizedType) {
            this.type = ((ParameterizedType)superClassType).getActualTypeArguments()[0];
        } else {
            throw new IllegalArgumentException("TypeReference는 항상 실제 타입 파라미터 정보가 있어야 합니다.");
        }
    }

    public Type getType() {
        return type;
    }
}
```

- TypeSafeMap: 수퍼 타입 토큰을 가지고 있는 TypeReference의 익명클래스를 파라미터로 받아 타입 안전성을 확보한 Map입니다.

TypeSafeMap

```
import java.lang.reflect.ParameterizedType;
import java.lang.reflect.Type;
import java.util.HashMap;
import java.util.Map;

public class TypeSafeMap {
    private final Map<Type, Object> map = new HashMap<>();  // key로 Type을 사용

    public <T> void put(TypeReference<T> k, T v) {
        map.put(k.getType(), v);
    }

    public <T> T get(TypeReference<T> k) {
        final Type type = k.getType();
        final Class<T> clazz;
        if (type instanceof ParameterizedType) {
            // 여기서 getRawType()을 통해 RawType을 가져오고 있습니다.
            clazz = (Class<T>) ((ParameterizedType) type).getRawType();
        } else {
            clazz = (Class<T>) type;
        }
        return clazz.cast(map.get(type));
    }
}
```

- getRawType()을 호출하면 RawType을 가져올 수 있습니다.
- RawType을 알기 위해 JLS 4.8의 문서를 보면 아래와 같은 문구가 있습니다.

> **"The superclasses (respectively, superinterfaces) of a raw type are the erasures of the superclasses (superinterfaces) of any of the parameterizations of the generic type."**

- 대략적인 뜻은 제네릭 타입이 모두 소거된 superclasses가 RawType이라는 것을 확인할 수 있습니다.
- 예를 들어 List 정보를 가지고 있는 ParameterizedType를 받아서 getRawType()으로 RawType을 가져오게 되면 그 값은 List.class 입니다.
- 이제 제네릭 정보가 모두 제거된 RawType을 통해 Type Casting을 할 수 있습니다!
- 자료형별 put과 get을 테스트하는 코드를 아래와 같이 작성했습니다.

TypeSafeMap 테스트

```
import java.util.Arrays;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.junit.jupiter.api.Test;

import main.super_type_token.TypeReference;
import main.super_type_token.TypeSafeMap;

import static org.junit.jupiter.api.Assertions.*;

class TypeSafeMapTest {
    private final TypeSafeMap typeSafeMap = new TypeSafeMap();

    @Test
    public void put_and_get_test_using_string() {
        final String inputData = "문자열";
        final TypeReference<String> tr= new TypeReference<>() {};

        // put
        typeSafeMap.put(tr, inputData);
        // get
        final String outputData = typeSafeMap.get(tr);

        System.out.println("input: " + inputData + "₩n" +  "output: " +outputData);
        assertEquals(inputData, outputData);
```

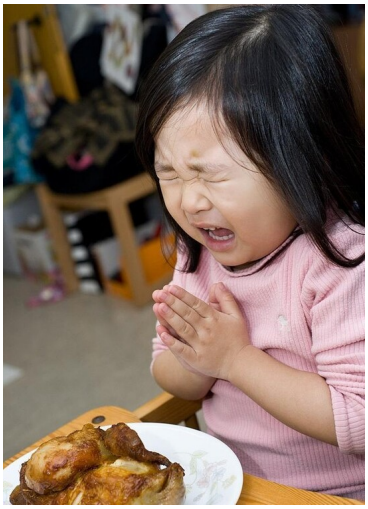
```
    }

    @Test
    public void put_and_get_test_using_integer() {
        final Integer inputData = 12345;
        final TypeReference<Integer> tr= new TypeReference<>() {};

        // put
        typeSafeMap.put(tr, inputData);
        // get
        final Integer outputData = typeSafeMap.get(tr);

        System.out.println("input: " + inputData + "₩n" +  "output: " +outputData);
        assertEquals(inputData, outputData);
    }

    @Test
    public void put_and_get_test_using_list_string() {
        final List<String> inputData = Arrays.asList("H", "O", "N", "G");
        // List<String>.class와 동일한 효과
        final TypeReference<List<String>> tr= new TypeReference<>() {};

        // put
        typeSafeMap.put(tr, inputData);
        // get
        final List<String> outputData = typeSafeMap.get(tr);

        System.out.println("input: " + inputData + "₩n" +  "output: " +outputData);
        assertEquals(inputData, outputData);
    }

    @Test
    public void put_and_get_test_using_list_list_string() {
        final List<List<String>> inputData = Arrays.asList(
            Arrays.asList("H", "O", "N", "G"),
            Arrays.asList("S", "U", "N", "G"),
            Arrays.asList("M", "I", "N")
        );
        // List<List<String>>.class와 동일한 효과
        final TypeReference<List<List<String>>> tr= new TypeReference<>() {};

        // put
        typeSafeMap.put(tr, inputData);
        // get
        final List<List<String>> outputData = typeSafeMap.get(tr);

        System.out.println("input: " + inputData + "₩n" +  "output: " +outputData);
        assertEquals(inputData, outputData);
    }

    @Test
    public void put_and_get_test_using_map() {
        final Map<String, String> inputData = new HashMap<>();
        inputData.put("key1", "value1");
        inputData.put("key2", "value2");
        inputData.put("key3", "value3");
        // Map<String, String>.class와 동일한 효과
        final TypeReference<Map<String, String>> tr= new TypeReference<>() {};

        // put
        typeSafeMap.put(tr, inputData);
        // get
        final Map<String, String> outputData = typeSafeMap.get(tr);

        System.out.println("input: " + inputData + "₩n" +  "output: " +outputData);
        assertEquals(inputData, outputData);
    }
}
```

너덜너덜.. 뽀개기 여정이 끝나가지만.. 아직도 슈퍼 타입 토큰을 사용하기 쉽지 않네요.. 더 더 더 쉬운걸 쓰고싶습니다!

5. 직접 구현하기에는 너무 복잡하네요.. 이미 잘 만들어진 라이브러리는 없을까요?

5_1. Spring을 쓰신다면 ParameterizedTypeReference를 사용해주세요!

- TypeReference을 직접 만들기 보다 Spring 흉님의 ParameterizedTypeReference를 사용해 보세요!



치느님!! 아.. 아니 스프링님!!

- Spring 프레임워크에서도 동일하게 런타임시 발생하는 타입 안정성 문제를 해결하기 위해 ParameterizedTypeReference라는 클래스를 만들었습니다. 클래스 네이밍을 생각해보면 ParameterizedType을 가진 Reference라는 뜻인 것 같습니다. 앞서 말씀드린 것 같이 ParameterizedType은 제네릭 타입을 가지고 있는 Type을 의미합니다. 즉 이 클래스는 제네릭을 가지고 있는 Type을 넘기고 싶을 때 사용하면 될 것 같습니다.
- Spring core 패키지에 있는 ParameterizedTypeReference는 아래와 같습니다.

ParameterizedTypeReference

```
/**
 * The purpose of this class is to enable capturing and passing a generic
 * {@link Type}. In order to capture the generic type and retain it at runtime,
 * you need to create a subclass (ideally as anonymous inline class) as follows:
 *
 * <pre class="code">
 * ParameterizedTypeReference<List<String>> typeRef = new ParameterizedTypeReference<List<String>>() {};
 * </pre>
 *
 * <p>The resulting {@code typeRef} instance can then be used to obtain a {@link Type}
 * instance that carries the captured parameterized type information at runtime.
 * For more information on "super type tokens" see the link to Neal Gafter's blog post.
 *
 * @author Arjen Poutsma
 * @author Rossen Stoyanchev
 * @since 3.2
 * @param <T> the referenced type
 * @see <a href="https://gafter.blogspot.nl/2006/12/super-type-tokens.html">Neal Gafter on Super Type Tokens</a>
 */
public abstract class ParameterizedTypeReference<T> {

	private final Type type;


	protected ParameterizedTypeReference() {
		Class<?> parameterizedTypeReferenceSubclass = findParameterizedTypeReferenceSubclass(getClass());
		Type type = parameterizedTypeReferenceSubclass.getGenericSuperclass();
		Assert.isInstanceOf(ParameterizedType.class, type, "Type must be a parameterized type");
		ParameterizedType parameterizedType = (ParameterizedType) type;
		Type[] actualTypeArguments = parameterizedType.getActualTypeArguments();
		Assert.isTrue(actualTypeArguments.length == 1, "Number of type arguments must be 1");
		this.type = actualTypeArguments[0];
	}

	private ParameterizedTypeReference(Type type) {
		this.type = type;
	}


	public Type getType() {
		return this.type;
	}

	@Override
	public boolean equals(@Nullable Object other) {
		return (this == other || (other instanceof ParameterizedTypeReference &&
				this.type.equals(((ParameterizedTypeReference<?>) other).type)));
	}

	@Override
	public int hashCode() {
		return this.type.hashCode();
	}

	@Override
	public String toString() {
		return "ParameterizedTypeReference<" + this.type + ">";
	}


	/**
	 * Build a {@code ParameterizedTypeReference} wrapping the given type.
	 * @param type a generic type (possibly obtained via reflection,
	 * e.g. from {@link java.lang.reflect.Method#getGenericReturnType()})
	 * @return a corresponding reference which may be passed into
	 * {@code ParameterizedTypeReference}-accepting methods
	 * @since 4.3.12
	 */
	public static <T> ParameterizedTypeReference<T> forType(Type type) {
		return new ParameterizedTypeReference<T>(type) {
		};
```

```
        }

        private static Class<?> findParameterizedTypeReferenceSubclass(Class<?> child) {
                Class<?> parent = child.getSuperclass();
                if (Object.class == parent) {
                        throw new IllegalStateException("Expected ParameterizedTypeReference superclass");
                }
                else if (ParameterizedTypeReference.class == parent) {
                        return child;
                }
                else {
                        return findParameterizedTypeReferenceSubclass(parent);
                }
        }

}
```

6. 그렇다면 수퍼 타입 토큰은 주로 어디에 사용할 수 있을까요?

6_1. RestTemplate을 사용할 때

- 다른 서버에 있는 자원을 가져오고 싶을 때 Client 라이브러리를 이용하는데요.
- 그 중에서 RestTemplate를 예를 들고 슈퍼 타입 토큰을 이용해 타입 안정성을 확보해보겠습니다.
- 우선 API 2개를 만들고 테스트 코드를 통해 검증해 보겠습니다. 시이작!

API 2개

```
import java.util.List;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }

    @RestController
    public static class MyController {
        @GetMapping("/users")
        public List<User> getUsers() {
            return List.of(new User("HONG", 10), new User("SUNG", 11), new User("MIN", 12));
        }

        @GetMapping("/products")
        public List<Product> getProducts() {
            return List.of(new Product("우유", 1000L), new Product("과자", 2000L), new Product("아이스크림", 3000L));
        }
    }

    public static class User {
        private String name;
        private int age;

        private User() { }

        public User(String name, int age) {
            this.name = name;
            this.age = age;
        }

        public String getName() {
            return name;
        }
```

```
        public int getAge() {
            return age;
        }

        @Override
        public String toString() {
            return "User{" +
                "name='" + name + '₩'' +
                ", age=" + age +
                '}';
        }
    }

    public static class Product {
        private String name;
        private long price;

        private Product() {
        }

        public Product(String name, long price) {
            this.name = name;
            this.price = price;
        }

        public String getName() {
            return name;
        }

        public long getPrice() {
            return price;
        }

        @Override
        public String toString() {
            return "Product{" +
                "name='" + name + '₩'' +
                ", price=" + price +
                '}';
        }
    }
}
```

- 테스트 코드를 작성합니다.
- 테스트 코드에서는 ParameterizedTypeReference를 익명클래스로 계속 만들어 사용중 이지만 실제로 사용하신다면 static하게 한번 만들고 재사용함 함으로써 성능을 향상시키면 좋을것 같습니다~!

ParameterizedTypeReference를 사용해 테스트하는 코드

```
import java.util.LinkedHashMap;
import java.util.List;

import org.junit.Assert;
import org.junit.Test;
import org.springframework.core.ParameterizedTypeReference;
import org.springframework.http.HttpMethod;
import org.springframework.web.client.RestTemplate;

import static com.example.demo.DemoApplication.*;

public class ParameterizedTypeReferenceUsageTest {
    private final RestTemplate restTemplate = new RestTemplate();
    private final static String USER_URI = "http://localhost:8080/users";
    private final static String PRODUCT_URI = "http://localhost:8080/products";

    @Test
    public void 슈퍼타입토큰X_user_api_호출해서_사용한_경우() {
        List<User> users = restTemplate.getForObject(USER_URI, List.class);
        final Object first = users.get(0);

        //User에 대한 타입토큰을 알 수 없어서 키벨류 형식을 변환할 수 있는 디폴트 자료형인 LinkedHashMap으로 변환을 했기 때문입니다.
        Assert.assertFalse(first instanceof User);
        Assert.assertTrue(first instanceof LinkedHashMap);
    }

    @Test
    public void 슈퍼타입토큰X__product_api_호출해서_사용한_경우() {
        List<User> users = restTemplate.getForObject(PRODUCT_URI, List.class);
        final Object first = users.get(0);

        //User에 대한 타입토큰을 알 수 없어서 키벨류 형식을 변환할 수 있는 디폴트 자료형인 LinkedHashMap으로 변환을 했기 때문입니다.
        Assert.assertFalse(first instanceof User);
        Assert.assertFalse(first instanceof Product);
        Assert.assertTrue(first instanceof LinkedHashMap);
    }

    @Test
    public void 슈퍼타입토큰O_user_api_호출해서_사용한_경우() {
        List<User> users = restTemplate.exchange(USER_URI,
                                                 HttpMethod.GET,
                                                 null,
                                                 new ParameterizedTypeReference<List<User>>() {})
                                       .getBody();
        final Object first = users.get(0);
        Assert.assertTrue(first instanceof User);

        users.forEach(System.out::println);
    }
}
```

6_2. ObjectMapper를 사용할 때

- 스프링에서 제공하는 ParameterizedTypeReference을 사용하지는 않습니다.
- 대신 TypeReference라는 Type을 멤버변수로 가진 추상클래스를 만들어 사용하고 있습니다. +) JavaType도 있지만 이는 밑에서 언급하겠습니다.
  또한 위에서 언급한 RestTemplate 에서도 필요시 ObjectMapper를 사용하고 있습니다.

ObjectMapper의 TypeReference 클래스

```
public abstract class TypeReference<T> implements Comparable<TypeReference<T>> {
    protected final Type _type;

    protected TypeReference() {
        Type superClass = this.getClass().getGenericSuperclass();
        if (superClass instanceof Class) {
            throw new IllegalArgumentException("Internal error: TypeReference constructed without actual type information");
        } else {
            this._type = ((ParameterizedType)superClass).getActualTypeArguments()[0];
        }
    }

    public Type getType() {
        return this._type;
    }

    public int compareTo(TypeReference<T> o) {
        return 0;
    }
}
```

- ObjectMapper의 readValue를 호출할 때 TypeReference를 익명클래스로 만들고 전달합니다.

ObjectMapper의 readValue

```
    /**
     * Method to deserialize JSON content from given JSON content String.
     *
     * @throws JsonParseException if underlying input contains invalid content
     *    of type {@link JsonParser} supports (JSON for default case)
     * @throws JsonMappingException if the input JSON structure does not match structure
     *   expected for result type (or has other mismatch issues)
     */
    public <T> T readValue(String content, TypeReference<T> valueTypeRef)
        throws JsonProcessingException, JsonMappingException
    {
        _assertNotNull("content", content);
        return readValue(content, _typeFactory.constructType(valueTypeRef));
    }
```

6_3. feign에서도 사용하나요..??

- 사내에서 Spring Cloud를 사용하고 있고 선언적으로 작성할 수 있는 clinet인 feign을 사용하고 있습니다.
- 혹시나 feign에서도 수퍼 타입 토큰을 사용하는지 알아보게 됐습니다. 제네릭 타입 또한 문제 없이 사용할 수 있었기 때문에 넘흐 궁금해졌습니다..
- 인터페이스를 통해 정의할 수 있는 **feign**도 **json response**를 받은 경우 데이터를 decode하기 위해 **jackson**의 **ObjectMapper**를 사용하고 있었습니다.
- **feign** 내 코드를 보면 아래 처럼 **Type**을 전달해주고 있는 것을 확인할 수 있습니다.

feign의 decode

```
Object decode(Response response, Type type) throws IOException {
    try {
      return decoder.decode(response, type);         //Type 줄게!
    } catch (final FeignException e) {
      throw e;
    } catch (final RuntimeException e) {
      throw new DecodeException(response.status(), e.getMessage(), response.request(), e);
    }
}
```

- **Type**을 전달받아 **JavaType**을 만들고 있었습니다.
- **ObjectMapper**는 **JavaType**을 전달 받아 **타입 캐스팅**할 수 있었습니다.

AbstractJackson2HttpMessageConverter.class

```
// AbstractJackson2HttpMessageConverter.class 내에서 javaType을 만들고 readJavaType 메서드로 전달 중
@Override
public Object read(Type type, @Nullable Class<?> contextClass, HttpInputMessage inputMessage)throws IOException,
HttpMessageNotReadableException {
		JavaType javaType = getJavaType(type, contextClass);
		return readJavaType(javaType, inputMessage);
}

// readJavaType메서드 코드내 ObjectMapper에게 어떤 자바 타입으로 변환해야 하는지 알리기 위해 JavaType을 전달 중
private Object readJavaType(JavaType javaType, HttpInputMessage inputMessage) throws IOException {
		try {
			if (inputMessage instanceof MappingJacksonInputMessage) {
				Class<?> deserializationView = ((MappingJacksonInputMessage) inputMessage).getDeserializationView();
				if (deserializationView != null) {
					return this.objectMapper.readerWithView(deserializationView).forType(javaType).
							readValue(inputMessage.getBody());
				}
			}
			return this.objectMapper.readValue(inputMessage.getBody(), javaType);
		}
		catch (InvalidDefinitionException ex) {
			throw new HttpMessageConversionException("Type definition error: " + ex.getType(), ex);
		}
		catch (JsonProcessingException ex) {
			throw new HttpMessageNotReadableException("JSON parse error: " + ex.getOriginalMessage(), ex, inputMessage);
		}
}
```

- Q: 오홋! 그렇다면 **feign** 도 **Type**을 넘겨주기 위해 당연히 **수퍼 타입 토큰**을 이용해야 겠네요!?

- A: ♂ 아닙니다!!!

- **feign**은 java.lang.reflect.**Method**를 이용해 메소드가 return할 **Type(제네릭 정보가 있는 ParameterizedType도 물론 포함)**을 가져올 수 있기 때문에 **수퍼 타입 토큰을 사용할 필요가 없습니다.**

class 내 method 에서 반환 Type을 가져오는 예제 코드

```
@Test
void Method를_통한_Type_가져오기() {
    Method method = StringLIstClient.class.getMethods()[0];
    Type type = method.getGenericReturnType();
    assertTrue(type instanceof ParameterizedType);
}
```

- 빈초기화 시기때 빈을 만들면서 리플렉션을 통해 **클래스내 메서드 정보**를 가져올 수 있고 이때 **Type** 또한 가져올 수 있습니다.
- 결국 **제네릭 타입 정보**를 가지고 있는 **ParameterizedType** 또한 가져올 수 있기 때문에 이를 **Jackson의 ObjectMapper** 등에 전달할 수 있었고 **안전**하게 **타입 캐스팅**이 가능했던 것입니다.

java.lang.reflect.Method 내 getGenericReturnType

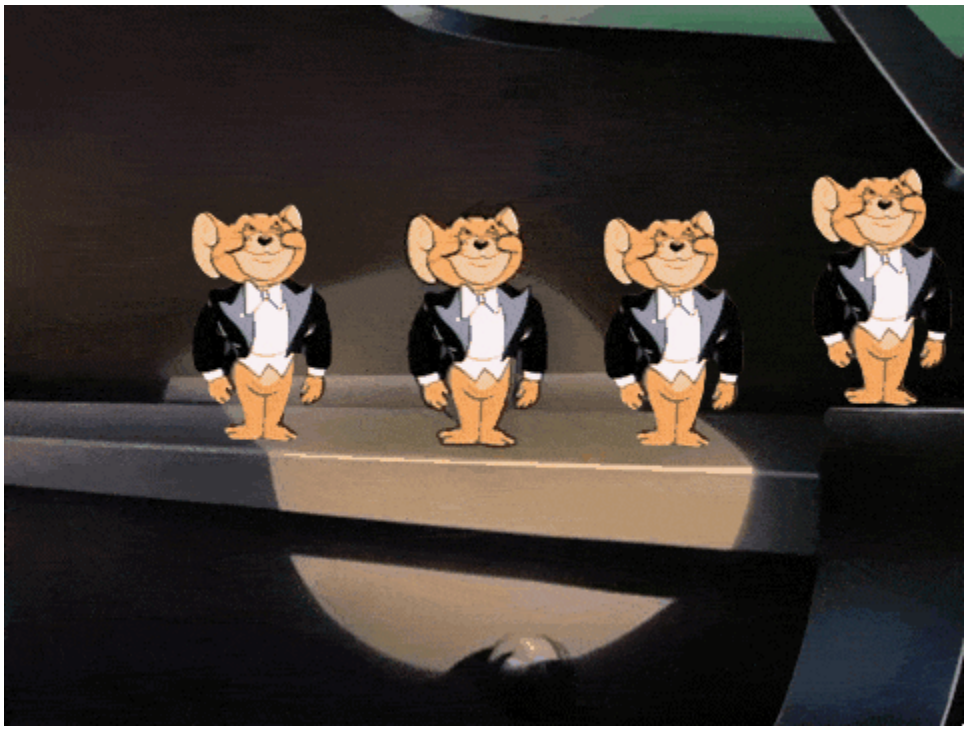
```
public Type getGenericReturnType() {
    return (Type)(this.getGenericSignature() != null ? this.getGenericInfo().getReturnType() : this.getReturnType());
}
```

결론

- 타입 안정성을 확보하기 위해서는 런타임 때도 제네릭 정보도 알 수 있어야 합니다.
- 하지만 타입 소거자에 의해 제네릭 타입은 사라집니다.
- 이를 극복하기 위한 편법으로 수퍼 타입 토큰을 이용합니다.
- 수퍼 타입을 이용하면 제네릭 타입을 넣을 수 있습니다.
- 많은 라이브러리에서 사용자에게 수퍼 타입 토큰을 담을 수 있는 추상클래스를 제공하고 있습니다.
- 사용자는 단순히 익명클래스에 제네릭 타입 정보를 넣어서 전달하면 됩니다.
- 그러면 ObjectMapper, Gson 같은 라이브러리에서 그 정보를 보고 타입 캐스팅을 하며 타입이 안맞을 경우 예외를 발생시킬 수 있습니다.
- 선언적인 프로그래밍이 가능한 feign은 클래스 내부의 메서드 정보에서 return되는 Object의 제네릭 정보도 가져올 수 있기 때문에 수퍼 타입 토큰이 필요 없습니다.

수퍼 타입 뽀개기 여정이 끝났습니다~!
오늘도 열심히 일하느라 당떨어진 자신에게 따듯한 선물을 주는 건 어떨까요?

긴 글 읽어주셔서 감사합니다! (_ _)



출처:

1. Neal Gafter's blog
2. HomoEfficio. 클래스 리터럴, 타입 토큰, 수퍼 타입 토큰
3. 토비의 봄 TV - 수퍼 타입 토큰

4. 토비의 봄 TV - 수퍼 타입 토큰(2), 스프링 ResolvableType ← 타입 정보를 알아낼 때 많은 예외처리가 필요합니다. 이런 경우 이미 잘 만들어진 ResolvableType을 사용하시는 것을 추천드립니다.