Consider typesafe heterogeneous containers

일반적으로 Collection에서 generic을 사용합니다.

ex:)

```
Set<E>, Map<K,V>, ThreadLocal<T>, AtomicReference<T>
```

이 예시들을 보면 container가 parameterized 됩니다!

```
Set<String> s;
```

라고 선언하면 변수 s가 container이고, 이 s는 String으로 parameterized되었습니다.

그래서 elements들이 해당 parameter의 type을 가지게 되는 형태죠.

하지만, 유연성을 줘야할 때가 있습니다. row에 임의적인 columns를 받는 database를 생각해보면, type safe한 방식으로 모든 열을 접근 할 수 있다면 좋겠지요

more flexible

idea

기존 방식대로 container를 Type parameterized하면 하나의 타입만 받을 수 있습니다. 하지만! container 대신 key를 parameterized한다면! 다양한 타입을 key값으로 둘 수 있게 되지요.

Example - Favorites

favorite class는 client가 다양한 타입의 인스턴스를 저장하고, 꺼낼 수 있도록 합니다.

구현 방법

- key: Class의 object를 parameterized key로 둡니다.
    - 이게 가능한 이유는 Class 클래스가 generic이기 때문이죠!

- Type.class literal

  그리고 Key는 Type.class를 이용하며 객체를 저장하면 됩니다. 이게 작동하는 이유는 `Type.class => Class<Type>`이기 때문이죠! 예를 들어서 `String.class => Class<String>`이 되는 것 이죠.

  > 한편, class literal이 method간 type information으로 파라미터로 넘겨질 경우 type token이라고 불립니다.

코드

Favorite API는 일반 Map과 비슷한 구조를 지닙니다. Map에서 Key가 parameterized된 형태이지요

API

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

  public <T> void putFavorite(Class<T> type, T instance) {
    favorites.put(Objects.requireNonNull(type), instance);
  }

  public <T> T getFavorite(Class<T> type) {
    return type.cast(favorites.get(type));
  }
}
```

클라이언트에서 사용 코드

```
// Typesafe heterogeneous container pattern - client
public static void main(String[] args) {
  Favorites f = new Favorites();
  f.putFavorite(String.class, "Java");
  f.putFavorite(Integer.class, 0xcafebabe);
  f.putFavorite(Class.class, Favorites.class);
```

```
    String favoriteString = f.getFavorite(String.class);
    int favoriteInteger = f.getFavorite(Integer.class);
    Class<?> favoriteClass = f.getFavorite(Class.class);
    System.out.printf("%s %x %s%n", favoriteString,
                      favoriteInteger, favoriteClass.getName());
    //output: Java cafebabe Favorites
}
```

> 참고로 자바의 %n은 C의 \n과 동일한 역할을 하나, 자바에서는 platform에 맞는 단어를 입력해줍니다.

구현 설명

1. Map과 unbounded wild card

favorites의 타입은 `Map<Class<?>, Object>`인데, 어떻게 다양한 타입을 넣을 수 있지? 할거에요. unbounded wild card가 쓰였으니까요. 그런데 다시 생각해보시면, container가 아닌, key에 unbounded wild card가 있기 때문에 다양한 타입을 넣을 수 있습니다.

2. Value

구현된 Map의 value를 보시면 단순히 Object type입니다. 즉, Key의 타입과 관련성이 없다는 의미죠. java의 type system은 key과 value의 관계를 표현할 수 있을 정도로 충분하지 않아서 어쩔 수 없습니다.

하지만, 관계가 올바르게 설정되어 있는 것을 elements를 꺼낼 때 확인할 수 있습니다.

putImple

단지 Class object type과 맞는 favorite instance를 넣습니다. 위에 서술한대로 집어넣을 때 둘 간의 *type linkage*가 끊어집니다.

getImple

favorites를 꺼내는 역할을 하는데, key와 value의 type관계가 다시 연결됩니다.

1. 주어진 Class object의 type과 맞는 value를 Map에서 꺼냅니다.

   반환하기에는 올바른 object reference이나, compile time엔 Object type으로 잘못된 타입을 가지고 있습니다. 따라서 Type T로 변환할 필요가 있죠.

2. Class의 cast method이용하여 type T로 dynamic casting을 합니다.

   cast method는 전달받은 argument의 type이 Class object에에 의해 표현된 타입 인스턴스인지 체크합니다.

체크한 뒤엔 해당 타입을 반환하죠. 체크 실패하면 ClassCastException을 던지고요.

집어넣었을 때 key에 맞는 value를 집어넣었기 때문에 ClassCastException이 발생하지 않는다.라고 말할 수 이습니다.

castMethod 동작방식

Class가 Generic이라는 점을 십분 이용한 메서드입니다. 코드는 다음과 같죠

```
public T cast(Object obj) {
  if (obj != null && !isInstance(obj))
    throw new ClassCastException(cannotCastMsg(obj));
  return (T) obj;
}
```

T로 unchecked cast할 필요없이 type safey를 보장할 수 있게 됩니다.

한계와 대안책

한계점 1

악의적인 클라이언트가 Raw type을 집어넣으면 문제가 됩니다.

예시

```
f.putFavorite((Class)Integer.class, "Integer의 인스턴스가 아닙니다.");
int favoriteInteger = f.getFavorite(Integer.class);
```

이는, HashSet<Integer>에서도 동일한 문제가 발생합니다.

```
HashSet<Integer> = new HashSet<>();
((HashSet)set).add("문자열 입니다.")
```

이 정도를 감수한다면 런타임 중 타입 안정성을 얻을 수 있습니다.

대안 - runtime type safety 구현

1) dynamic casting 활용

```
public <T> void putFavorite(Class<T> type, T instance) {
  favorites.put(type, type.cast(instance));
}
```

2) Collections에서 지원하는 dynamic casting사용

Collections에 있는 checkedSet, checkedList, checkedMap이 위 방식을 적용한 collection wrapper입니다.

```
public static <E> Set<E> checkedSet(Set<E> s, Class<E> type) {
  return new CheckedSet<>(s, type);
}


static class CheckedSet<E> extends CheckedCollection<E>
  implements Set<E>, Serializable
{
  private static final long serialVersionUID = 4694047833775013803L;

  CheckedSet(Set<E> s, Class<E> elementType) { super(s, elementType); }

  public boolean equals(Object o) { return o == this || c.equals(o); }
  public int hashCode()           { return c.hashCode(); }
}
```

```
static class CheckedCollection<E> implements Collection<E>, Serializable {
  private static final long serialVersionUID = 1578914078182001775L;

  final Collection<E> c;
  final Class<E> type;

  @SuppressWarnings("unchecked")
  E typeCheck(Object o) {
    if (o != null && !type.isInstance(o))
      throw new ClassCastException(badElementMsg(o));
    return (E) o;
  }
```

이 래퍼클래스들은 실체화시키며 잘못된 타입을 집어넣으려고 하면 ClassCastException을 던집니다. 그래서 제너릭과 로 타입을 섞어 쓰는 어플리케이션에서 클라이언트 코드가 컬렉션에 잘못된 타입의 원소를 넣지 못하게 추적하는데 도움을 주죠.

한계점 2

non-reifiable type엔 사용할 수 없습니다.

`List<String>`을 예시로 들죠.

`List<String>.class`를 사용할 수 없습니다. 만약에 허용한다면 `List<String>.class`와 `List<Integer>.class`를 입력할겁니다.

하지만 이들은 runtime중엔 type erase되어 List.class로 동일한 object를 가르키게 될 것이니 문제가 되겠죠?

대안 -

이 한계점에선 완벽히 만족할만한 대안은 없습니다.

> 옮긴이의 글 - super type token
>
> http://bit.ly/2NGQi2S 참조.

Unbounded type token

Favorites에서 사용하는 타입토큰은 unbounded type token이지만 상황에 따라 bounded type을 사용하고 싶을 수 있습니다.

여기서 *bounded type token*이란, 단지 bounded type parameter나 bounded wildcard를 이용해서 표현 타입을 제한하는 type token을 의미합니다.

예시 - AnnotationAPI

AnnotationAPI는 적극적으로 bounded type token을 사용합니다!

아래처럼 Runtime에 annotation을 읽는 메서드가 있습니다.

```
public <T extends Annotation>
            T getAnnotation(Class<T> annotationType);
```

이 method는 AnnotatedElement의 Interface의 method이며 class, method, fields등 다른 프로그램의 요소를 나타내는 reflective types에 의해 구현됩니다. (reflective types란 reflection 대상이 되는 type을 일컫습니다. ex: java.lang.Class<T>, java.lang.reflect.Method, java.lang.reflect.Field)

여기서, annotationType argument가 바로 annotation의 타입을 나타내는 bounded type token입니다.만약에 타입을 가지고 있다면 해당 타입을 반환할 것이며 아니라면 null을 반환할 겁니다.

결국, annotated element는 key가 annotation types인 heterogeneous typesafe container가 됩니다.

Class<?>와 asbSubclass

Class<?> type token으로 넘기기

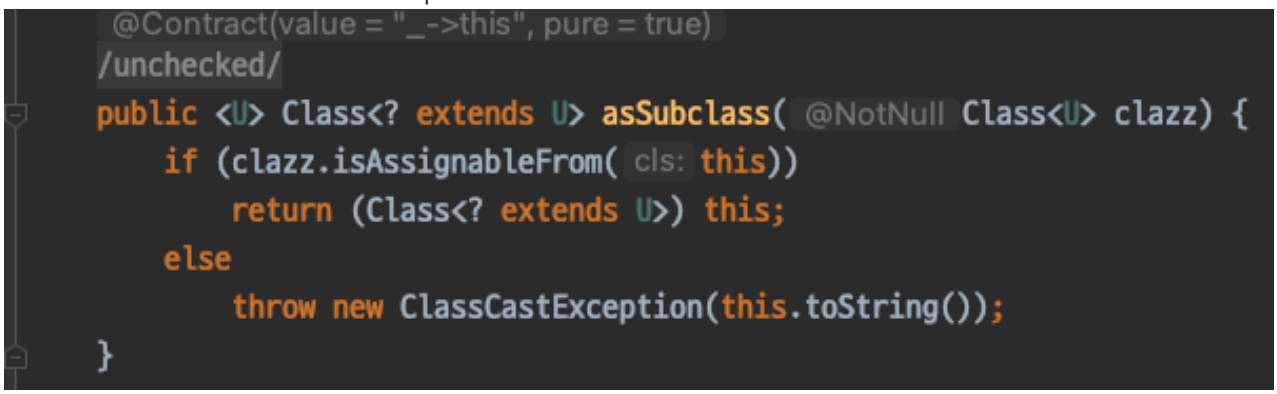
만약에 Class<?>타입의 객체가 있고 한정적 타입 토큰을 받는 메서드에 값을 넘기려면 어떻게 해야할까요?

object를 Class<? extends Annotation>으로 형변환할 수도 있긴하지만 unchecked warning이여서 컴파일 경고가 발생할 수 있습니다.

asSubclass

이 형변환을 안전하고 동적으로 수행해주는 메서드가 바로 asSubclass입니다.

해당 argument에 표현된 subclass를 나타내기 위해 Class object가 캐스팅합니다. 성공시 인수로 받는 클래스 객체를 반환하고 아니면 ClassCastException을 던지죠.



asSubclass method를 이용하여 compile time에 type을 모르는 annotation을 어떻게 읽는지 다음 코드로 보여드리겠습니다.

```
// Use of asSubclass to safely cast to a bounded type token
static Annotation getAnnotation(AnnotatedElement element,
                                String annotationTypeName) {
  Class<?> annotationType = null; // Unbounded type token try {
  annotationType = Class.forName(annotationTypeName);
} catch (Exception ex) {
  throw new IllegalArgumentException(ex);
}
return element.getAnnotation( annotationType.asSubclass(Annotation.class));
}
```