Generic Functions in Java

PAva Project 1
Group 7
Daniel Correia, 80967 MEIC-A

Motivations

1. Use reflection only when it’s absolutely necessary
   a. Exceptions are annoying to deal with
   b. Performance overhead vs normal method calls

Performance Overhead

Because reflection involves types that are dynamically resolved, certain Java virtual machine optimizations can not be performed. Consequently, reflective operations have slower performance than their non-reflective counterparts, and should be avoided in sections of code which are called frequently in performance-sensitive applications.

(quote from the Java docs on Reflection)

Motivations

2. It should be possible to know the state of the dispatcher at any moment
   a. For example, how many generic functions are available?
   b. Why? Debugging, reusability, logging, etc…
   c. Consequence? Need a data structure instead of a “silver bullet” method

```
static Method bestMethod(Class type, String name, Class argType)
    throws NoSuchMethodException {
    try {
        return type.getMethod(name, argType);
    } catch (NoSuchMethodException e) {
        if (argType == Object.class) {
            throw new NoSuchMethodException(name);
        } else {
            return bestMethod(type,
                              name,
                              argType.getSuperclass());
        }
    }
}
```

Solution - Mapception

Nested HashMap containing information on available GenericFunction methods (before, primary, after).

Each map has 3 levels using different keys as parameters to support several features:

Level 1 (Class, HashMap): support multiple GenericFunction annotated classes

Level 2 (String, HashMap): support multiple method names inside a GenericFunction annotated class.

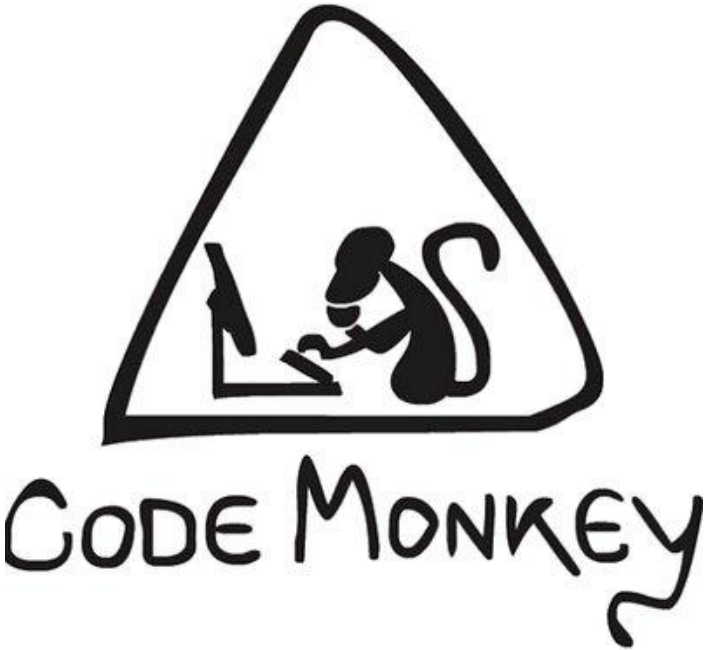
Level 3 (String, Method): support multiple parameters types combinations for the same method name.

Solution - Mapception

```
someMap = {
    GFClass = {
        methodName = {
                paramsId = Method object,
        }
    }
}
```

Solution - Mapception (example)

```
gFunPrimaryMap = {
    interface examples.enunciado.Com = {
        bine = {
            java.lang.String#java.lang.Integer = Object Com.bine(String,Integer),
            java.lang.Integer#java.lang.Integer = Integer Com.bine(Integer,Integer),
            java.lang.String#java.lang.Object = Object Com.bine(String,Object),
            java.lang.Object#java.lang.Object = Object Com.bine(Object,Object)
        }
    }
}
```



The rest is just coding work...

Translator

Replace GenericFunction method calls

```
if (m.getMethod().getDeclaringClass().hasAnnotation(GenericFunction.class)){
    m.replace(String.format(
        "{ $_ = ($r) GenericFunctionDispatcher.invokeGenericFunction($class, \"%s\", ($args)); }"
            , m.getMethodName()));
}
```

Initialize Map by Annotation

Input: target Class, annotation Class (e.g BeforeMethod.class)

1. For each Method in getDeclaredMethods
2. If annotation is present
   a. method.setAccessible(true)
   b. Build method parameters id
   c. Update result map

```
List<String> paramsClassNames = Lists.transform(Arrays.asList(method.getParameterTypes()), Class::getName);
```

```
// Method didn't exist yet -> create map for the method and add the new parameters types map
clazz_methods.put(method.getName(), Maps.newHashMap(ImmutableMap.of(paramsId, method)));
```

Parameters Combinations

Handled by a CombinationsHelper module

1. Convert params Object[] to List<Class>
2. Build List<List<Class>> where a sublist is the class tree of a parameter (interfaces+superclasses)
    - Example: two parameters String and Integer
    - returns [[java.lang.String, interface Comparable, ...], [Integer, ... , Number, ..., Object]]
3. Generate combinations of sublists

Parameters Combinations

Dispatcher uses this to generate parameters ids combinations

```
private static List<String> getParamsIdCombinations(Object[] params, boolean leastToMostSpecific) {
    // Get all possible id combinations for the input parameters
    List<String> paramsIdCombinations = CombinationsHelper.getSuperCombinations(params).stream()
            .map(combination -> Lists.transform(combination, Class::getName))
            .map(GenericFunctionDispatcher::buildParamsId)
            .collect(Collectors.toList());
    // Reverse order of precedence list if we want least to most specific ordering (e.g. after methods)
    return leastToMostSpecific ? Lists.reverse(paramsIdCombinations) : paramsIdCombinations;
}
```

Find Primary Method

```
private static Method getPrimaryMethod(HashMap<String, Method> primaryParamsMap, Object[] params) throws NoSuchMethodException {
        return getParamsIdCombinations(params, false).stream()
                .filter(primaryParamsMap::containsKey).findFirst().map(primaryParamsMap::get)
                .orElseThrow(() -> new NoSuchMethodException("Unable to find a primary method during multiple dispatch"));
}
```

1. Get parameters ids combinations with most-to-least specific ordering (flag=false)
2. Filter for combinations that exist in the primary functions map
3. Get the first method after filtering
4. Otherwise, the user made a mistake and no primary method is available (exception)

Invoke Applicable Methods

Same logic as getPrimaryMethod but instead of findFirst, invoke all valid methods.

```
private static void invokeApplicableMethodsFromMap(HashMap<String, Method> paramsMap,
        boolean leastToMostSpecific, Object[] args){
    getParamsIdCombinations(args, leastToMostSpecific).stream()
            .filter(paramsMap::containsKey).map(paramsMap::get)
            .forEach(method -> invokeMethod(method, args));
}
```

Invoke Generic Function (1)

Initialize maps if it’s the first time a Generic Function class is used

```
public static Object invokeGenericFunction(Class clazz, String name, Object[] args) throws NoSuchMethodException {
    if (!gFunPrimaryMap.containsKey(clazz)) {
        // If it's the first time the dispatcher sees this class,
        //   then initialize all methods maps for this class (before, primary, after)
        gFunPrimaryMap.put(clazz, getInitMapForAnnotation(clazz, null));
        gFunBeforeMap.put(clazz, getInitMapForAnnotation(clazz, BeforeMethod.class));
        gFunAfterMap.put(clazz, getInitMapForAnnotation(clazz, AfterMethod.class));
    }
    ...
}
```

Invoke Generic Function (2)

```
public static Object invokeGenericFunction(Class clazz, String name, Object[] args) throws NoSuchMethodException {
    ...
    // Call applicable before methods (order: most specific to least specific)
    Optional.ofNullable(gFunBeforeMap.get(clazz).get(name))
            .ifPresent(beforeParamsMap -> invokeApplicableMethodsFromMap(beforeParamsMap, false, args));

    // Call most specific primary method
    Method primaryMethod = getPrimaryMethod(gFunPrimaryMap.get(clazz).get(name), args);
    Object invocation_result = invokeMethod(primaryMethod, args);

    // Call applicable after methods (order: least specific to most specific)
    Optional.ofNullable(gFunAfterMap.get(clazz).get(name))
            .ifPresent(afterParamsMap -> invokeApplicableMethodsFromMap(afterParamsMap, true, args));

    return invocation_result;
}
```

Extensions

- Effective Method Caching
  - Upgrade Mapception data structure
  - Difficulty: Easy
- Around Methods
  - New annotation
  - Placeholder “call-next-method” to be replaced with Javassist
  - New map for around methods
  - Handling multiple around methods + call-next-method logic
  - Difficulty: ??

Effective Method Caching

```
gFunPrimaryMap = {
    GFClass = {  methodName = {
        paramsId = {
            "cached" = {
                "before": List of Methods,
                "base": Primary Method,
                "after": List of Methods
            },
            "method" = Method object
        }
}}}
```

Effective Method Caching

1. Find most specific primary method (a map with keys “cached” , “method”)
2. If “cached” map is empty, do default behavior using Method in “method” and update “cached” with invoked methods.
3. Otherwise, get methods from “cached” map

Around Method - Annotation

New Annotation

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface AroundMethod {
}
```

Around Method - Marker

Provide to user a call-next-method placeholder function

```
public class WithGenericFunction {
    ...
    public static Object callNextMethod(Object... args){
            // Do nothing. This method is used as a marker in @AroundMethods to
            // replace with a javassist method that handles call-next-method functionality
            return null;
    }
}
```

Around Method - Example

```
@GenericFunction
interface VoidColor {
    public static void doThings(C1 c){
        System.out.println("C1");
    }

    @AroundMethod
    public static void doThings(Foo c){
        System.out.println("Foo Start");
        WithGenericFunction.callNextMethod(c);
        System.out.println("Foo End");
    }
}
```

Around Method - Translator

Upgrade Translator to replace “call-next-method” markers

```
if (ctMethod.hasAnnotation(AroundMethod.class) && m.getMethodName().equals("callNextMethod")){
    m.replace(String.format("{ $_ = ($r) AroundMethodHelper.callNextMethod(%s.class, \"%s\", ($args)); }"
    , ctClass.getName(), ctMethod.getName()));
}
else if (m.getMethod().getDeclaringClass().hasAnnotation(GenericFunction.class)) {
    ...
}
```

Around Method - Dispatcher

1. Add around method map initialization
2. Use AroundMethodHelper to try to invoke an around method, if available

```
public static Object invokeGenericFunction(Class clazz, String name, Object[] args)
     throws NoSuchMethodException {
    if (!gFunPrimaryMap.containsKey(clazz)) {
        gFunPrimaryMap.put(clazz, getInitMapForAnnotation(clazz, null));
        gFunBeforeMap.put(clazz, getInitMapForAnnotation(clazz, BeforeMethod.class));
        gFunAfterMap.put(clazz, getInitMapForAnnotation(clazz, AfterMethod.class));
        AroundMethodHelper.initializeAroundMethodMap(clazz);
    }
    return AroundMethodHelper.tryInvokeAroundMethod(clazz, name, args);
}
```

Around Method - Try to Invoke

Keep track of which around methods have already been invoked in the current invocation context to avoid infinite recursion.

If not null, invoke the around method. Otherwise, invoke applicable methods.

```
private static HashMap getAroundMethodBySuperClasses(HashMap<String, HashMap> paramsMap, Object[] params) {
    for (String paramsId : GenericFunctionDispatcher.getParamsIdCombinations(params, false)) {
        if (paramsMap.containsKey(paramsId) && !usedAroundMethods.contains(paramsId)){
            HashMap methodCacheMap = paramsMap.get(paramsId);
            usedAroundMethods.add(paramsId);
            return methodCacheMap;
        }
    }
    return null;
}
```

Around Method - Call Next Method

```
public static Object callNextMethod(Class clazz, String name, Object[] args) throws NoSuchMethodException {
        Object[] actual_args = (Object[]) args[0]; // Necessary since Javassist wraps the args twice...
        HashMap<String, HashMap> aroundParamsMap = gFunAroundMap.get(clazz).get(name);
        HashMap aroundMethodMap = getAroundMethodBySuperClasses(aroundParamsMap, args);

        if (aroundMethodMap != null && !aroundMethodMap.isEmpty()) {
            // if i found an around method, then i remove it from used list before going recursively with invokeGenericFunction
            // this is necessary since the getAroundMethod call will be done again in the recursive call in tryInvokeAroundMethod
            usedAroundMethods.remove(usedAroundMethods.size()-1);
            Object result = GenericFunctionDispatcher.invokeGenericFunction(clazz, name, actual_args);
            usedAroundMethods.remove(usedAroundMethods.size()-1);
            return result;
        }
        else {
            Object result = GenericFunctionDispatcher.invokeApplicableMethods(clazz, name, actual_args);
            usedAroundMethods.remove(usedAroundMethods.size()-1);
            return result;
        }
    }
```

Questions?