Generic Functions

Group 01

Main Goal

Implement Generic Functions in Java

According to CLOS semantics

What it does?

- Standard method combination
- Auxiliary methods support
    - Before
    - After

What it doesn’t?

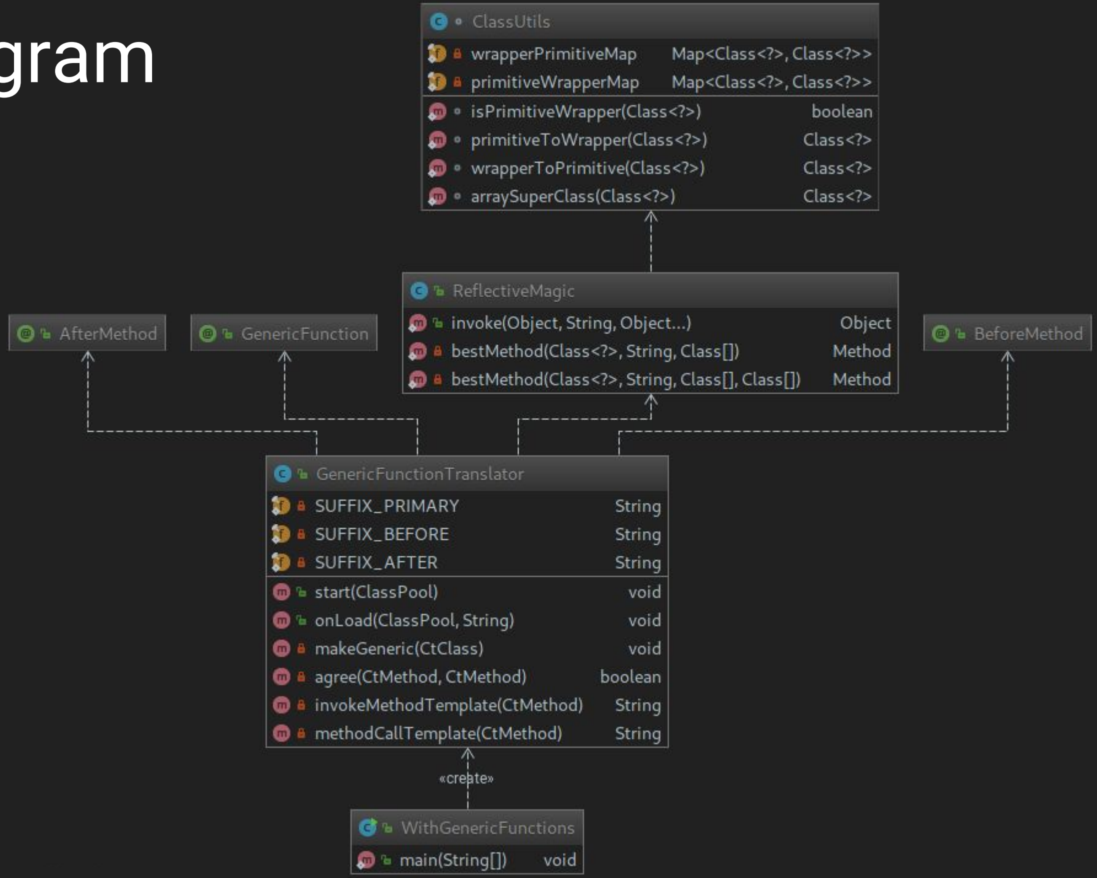
- The rest…
  - Around methods
  - Other method combinations
    - List
    - And
    - Or
    - …

But how?

Is it magic?

Yes!.. Kind of.. Not really

Class Diagram



ClassUtils	
wrapperPrimitiveMap	Map<Class<?>, Class<?>>
primitiveWrapperMap	Map<Class<?>, Class<?>>
isPrimitiveWrapper(Class<?>)	boolean
primitiveToWrapper(Class<?>)	Class<?>
wrapperToPrimitive(Class<?>)	Class<?>
arraySuperClass(Class<?>)	Class<?>

ReflectiveMagic	
invoke(Object, String, Object...)	Object
bestMethod(Class<?>, String, Class[])	Method
bestMethod(Class<?>, String, Class[], Class[])	Method

AfterMethod

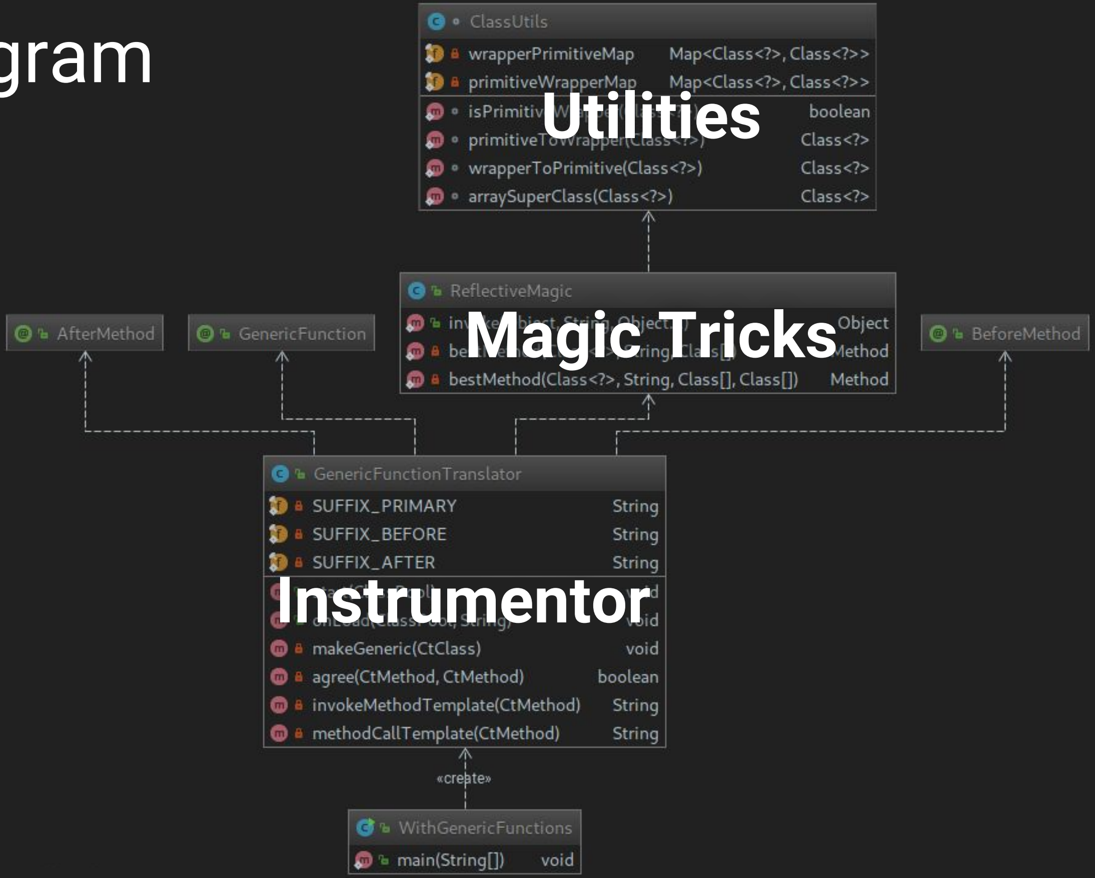
GenericFunction

BeforeMethod

GenericFunctionTranslator	
SUFFIX_PRIMARY	String
SUFFIX_BEFORE	String
SUFFIX_AFTER	String
start(ClassPool)	void
onLoad(ClassPool, String)	void
makeGeneric(CtClass)	void
agree(CtMethod, CtMethod)	boolean
invokeMethodTemplate(CtMethod)	String
methodCallTemplate(CtMethod)	String

WithGenericFunctions	
main(String[])	void

Class Diagram

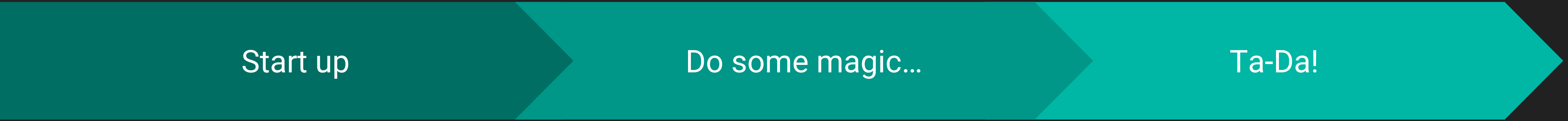

ClassUtils
wrapperPrimitiveMap Map<Class<?>, Class<?>>
primitiveWrapperMap Map<Class<?>, Class<?>>
isPrimitiveWrapper(Class<?>) boolean
primitiveToWrapper(Class<?>) Class<?>
wrapperToPrimitive(Class<?>) Class<?>
arraySuperClass(Class<?>) Class<?>
ReflectiveMagic
invoke(Object, String, Object[]) Object
bestMethod(Class<?>, String, Class[]) Method
bestMethod(Class<?>, String, Class[], Class[]) Method
AfterMethod
GenericFunction
BeforeMethod
GenericFunctionTranslator
SUFFIX_PRIMARY String
SUFFIX_BEFORE String
SUFFIX_AFTER String
start(ClassPool) void
onLoad(ClassPool, String) void
makeGeneric(CtClass) void
agree(CtMethod, CtMethod) boolean
invokeMethodTemplate(CtMethod) String
methodCallTemplate(CtMethod) String
«create»
WithGenericFunctions
main(String[]) void

Utilities

Magic Tricks

Instrumentor

How do?



Start up

- Run your program with `WithGenericFunctions`
- Creates a `Loader`
- Adds a `Translator`
- `run`s the program

Do some magic…

- Classes get loaded
- Intercept classes with `@GenericFunction`
- `makeGeneric` (More on that in a few)

Ta-Da!

- That’s it! Ta-da!
- Now cry because it isn’t as good as CLOS

HALT!

“Uninteresting” code snippets will be cut off. If you really want to see them… Save it for later, if we have time!

Start up

```
public class WithGenericFunctions {

 public static void main(String[] args) throws Throwable {
   if (args.length < 1) {
     System.err.printf("Usage: java %s <fq class name> [args]%n",
         WithGenericFunctions.class.getSimpleName());
     System.exit(1);
   } else {
     Loader loader = new Loader();
     loader.addTranslator(ClassPool.getDefault(), new GenericFunctionTranslator());
     String[] rest = new String[args.length - 1];
     System.arraycopy(args, 1, rest, 0, rest.length);
     loader.run(args[0], rest);
   }
 }
}
```

Interception (don’t mistake with intercession ;))

```java
public class GenericFunctionTranslator implements Translator {
  ...
  @Override
  public void onLoad(ClassPool pool, String className) throws ... {
    CtClass target = pool.get(className);
    if (target.hasAnnotation(GenericFunction.class)) {
      makeGeneric(target);
    }
  }

  private void makeGeneric(CtClass target) throws ... { ... }
  ...
}
```

Transformation

```
private void makeGeneric(CtClass target) throws ... {
  CtMethod[] methods = target.getDeclaredMethods();
  Arrays.sort(methods, (m1, m2) -> agree(m1, m2) ? 1 : -1); // Least specific first

  List<CtMethod> befores = new LinkedList<>();
  List<CtMethod> primaries = new LinkedList<>();
  List<CtMethod> afters = new LinkedList<>();

  for (CtMethod method : methods) {
    CtMethod newMethod = CtNewMethod.copy(method, method.getDeclaringClass(), null);
    Distinguish and rename original methods
    Inject 'invoke' into new method's body and add it to the class
  }
  For every primary, insert every 'before' and 'after' methods
}
```

Methods renamed

```java
@GenericFunction
public interface Explain {

  public static void it(Integer i) { ... }

  public static void it(Double i) { ... }

  public static void it(String s) { ... }

  @BeforeMethod
  public static void it(Number n) { ... }

  @AfterMethod
  public static void it(Object o) { ... }
}
```

```java
@GenericFunction
public interface Explain {

  public static void it$primary(Integer i) { ... }

  public static void it$primary(Double i) { ... }

  public static void it$primary(String s) { ... }

  @BeforeMethod
  public static void it$before(Number n) { ... }

  @AfterMethod
  public static void it$after(Object o) { ... }
}
```

Copies w/ *invoke*

```
@GenericFunction
public interface Explain {
  ...
  public static void it(Integer i) {
    ReflectiveMagic.invoke(Explain.class, "it$primary", new Object[]{i});
  }

  public static void it(Double i) { ... }

  public static void it(String s) { ... }

  public static void it(Number n) { ... }

  public static void it(Object o) { ... }
  ...
}
```

Befores and Afters

```
@GenericFunction
public interface Explain {
  ...
  public static void it$primary(Integer i) {
    if (i instanceof Number) {
      it$before((Number)i);
    }

    System.out.print(i + " is an integer");

    if (i instanceof Object) {
      it$after((Object)i);
    }
  }
  ...
}
```

Now.. MAGIC!

Though a magician never reveals his tricks.. We’ll make an exception

Well.. It clearly isn’t magic..

Just...

Painful code...

Praise Java! \(x.x)/

bestMethod does some “magic”

```
public class ReflectiveMagic {

  @SuppressWarnings("unused") // As a matter of fact, it is!
  public static Object invoke(Object receiver, String name, Object... args) {
    Class<?>[] argTypes = Arrays.stream(args).map(Object::getClass).toArray(Class[]::new);
    if (!(receiver instanceof Class<?>)) {
      receiver = receiver.getClass();
    }
    try {
      Method method = bestMethod(((Class<?>) receiver), name, argTypes);
      return method.invoke(receiver, args);
    } catch (ReflectiveOperationException roe) {
      Print out error message
      System.exit(1);
    }

    return null; // Never reached
  }

  private static Method bestMethod(Class<?> type, String name, Class[] argTypes) throws ... {
    return bestMethod(type, name, argTypes.clone(), argTypes);
  }
}
```

It really does!.. not...

```
private static Method bestMethod(Class<?> type, String name, Class[] origTypes, Class[] argTypes)
   throws NoSuchMethodException {
 try {
   return type.getMethod(name, argTypes);
 } catch (NoSuchMethodException nsme) {
   Check where the last Object is
   if (current == Object.class) {
     If it's the first, it means a method couldn't be found
     Otherwise, proceed to the previous argument
   } else {
     Check if it is an array, primitive or wrapper and convert it
     If a conversion happened, try getting the method with the converted type
     Otherwise, try crawling the interface hierarchy
   }
   Ultimately, if all else failed, crawl up class hierarchy
 }
}
```

Let’s expand (some of) those

Because those are “interesting”

Check where the last Object is

```
int i = argTypes.length - 1;
while (i > 0 && argTypes[i] == Object.class) {
  argTypes[i] = origTypes[i--]; // reset the current type
}
```

Check if it is an array, primitive or wrapper and convert it

```
Class<?> converted = current;
if (current.isArray()) {
  converted = ClassUtils.arraySuperClass(current);
} else if (current.isPrimitive()) {
  converted = ClassUtils.primitiveToWrapper(current);
} else if (ClassUtils.isPrimitiveWrapper(current)) {
  converted = ClassUtils.wrapperToPrimitive(current);
}
```

Otherwise, try crawling the interface hierarchy

```
for (Class<?> iface : current.getInterfaces()) {
  argTypes[i] = iface;
  Method method = bestMethod(type, name, origTypes, argTypes);
  if (method != null) {
    return method;
  }
}
```

Lots of code

Take a breather, no more ahead!
Well.. sort of

Just.. a few more things

Type tests

- Type I tests are all OK ✓
- Type II tests
  - TestE - Private (unaccessible) methods ⇒ Aren’t called
  - TestF - Interface order ⇒ Followed
  - TestJ - No applicable method ⇒ Reported
  - TestO - Method with Before & After ⇒ Effectively calls method 2x appropriately

TestE

```
public class TestE {
 public static void main(String[] args) {
   Object objects = new Object[]{123, "Foo", 1.2};
   System.out.println(Identify.it(objects));
 }
}
```

```
ObjectStringObject
```

```
@GenericFunction
public class Identify {
  public static String it(Object o) { return "Object"; }
  public static String it(String s) { return "String"; }
  private static String it(Integer a) { return "Integer"; }

  public static String it(Object[] arr) {
    StringBuilder res = new StringBuilder();
    for (Object o : arr) {
      res.append(it(o));
    }
    return res.toString();
  }
}
```

TestF

```
public class TestF {
 public static void main(String[] args) {
   Object c1 = new C1(), c2 = new C2();
   Bug.bug(c1);
   Bug.bug(c2);
 }
}
```

↓

```
Foo
Bar
```

```
public class C1 implements Foo, Bar {}
public class C2 implements Bar, Foo {}

@GenericFunction
public class Bug {
 public static void bug(Object o) { System.out.println("Object"); }

 public static void bug(Foo f) { System.out.println("Foo"); }

 public static void bug(Bar b) { System.out.println("Bar"); }
}
```

TestJ

```
public class TestJ {
 public static void main(String[] args) {
   Color blue = new Blue();
   What.is(blue);
 }
}
```

```
No applicable method: When calling
'What.is' with (Blue), no method is
applicable.
No restarts available (this isn't as good
as CLOS)
```

```
@GenericFunction
public class What {
 @BeforeMethod
 public static void is(Blue o) { ... }

 public static void is(Black i) { ... }
 public static void is(Red i) { ... }

 @AfterMethod
 public static void is(Object o) { ... }
 @AfterMethod
 public static void is(Color o) { ... }
 @AfterMethod
 public static void is(SuperBlack o) { ... }
}
```

TestJ

```java
public class TestO {
 public static void main(String[] args) {
   Object c = new C1();
   MakeIt.ddouble(c);
 }
}
```

```
Foo
Object
C1
Object
Foo
```

```java
@GenericFunction
public class MakeIt {
 public static void ddouble(C1 c) {
   System.out.println("C1");
 }

 @BeforeMethod
 @AfterMethod
 public static void ddouble(Object c) {
   System.out.println("Object");
 }

 @BeforeMethod
 @AfterMethod
 public static void ddouble(Foo c) {
   System.out.println("Foo");
 }
}
```

Pitfalls

- EVERY primary has EVERY before and after method
  - P primaries, B before methods, A after methods and C parameters
  - $\#\text{instanceof} \simeq P * C * (A + B)$
  - Same number of casts is made
- No support for generic functions w/ primitive parameter types
- *`bestMethod`* wasn’t tested exhaustively
  - Might fail in some cases

That’s it!

Any questions?