# Tips and Tricks used in the implementation

## ThreadLocal

In Java, variables are lexically scoped but sometimes you want to create *dynamic* variable like a current user or a current transaction. The class ThreadLocal represents thread local variables.

It can be seen as a `Map<Thread, Object>` which has a different values for each thread. `threadLocal.set(value)` set the value for the current thread, `threadLocal.get()` returns the value for the current thread and `threadLocal.remove()` removes the value associated with the current thread.

In order to avoid memory leaks, i.e. an object not being garbage collected because it is still stored in a thread local variable, we "scope" the variable to the stack so the thread local variable value is removed when the execution of the method ends.

```
private static final ThreadLocal<Data> DATA_THREAD_LOCAL = new ThreadLocal<>();

void method() {
  Data data = ...
  DATA_THREAD_LOCAL.set(data);
  try {
    ...
  } finally {
    DATA_THREAD_LOCAL.remove();
  }
}
```

In a close future, there is a proposal to introduce a new class ScopeLocal which is faster and always biased to the stack compared to a `ThreadLocal`.

## ClassValue

A ClassValue is a cache that allows to store information for a `Class`. It can be seen as the equivalent of a `Map<Class,Object>` but guarantee that classes can be unloaded.

The API works like a cache, the method `get(Class<?>)` try to retrieve the value from the `ClassValue`, if the value has not be computed, the `ClassValue` calls the method `computeValue` to get the value that will be stored in the cache for that `Class`.

```
private static final ClassValue<DATA> DATA_CLASS_VALUE = new ClassValue<>() {
    @Override
    protected Data computeValue(Class<?> type) {
      Data data = ...
      return data;
    }
  };

void method(Class<?> type) {
  Data data = DATA_CLASS_VALUE.get(type);
```

```
    ...
  }
```

# Default values

Sometimes you want to get the default values ( `null` , `false` , `0` , `0.0` , etc) of a `Class` . The default values is always `null` apart if the class is a primitive type. Primitive types are represented by a one letter descriptor in the classfile format (the .class file), accessible using the method Class.descriptorString().

```
private static final Map<String, Object> DEFAULT_VALUES = Map.of(
      "Z", false, "B", (byte) 0, "C", '\0', "S", (short) 0, "I", 0, "J", 0L, "F", 0f, "D", 0.);

Object defaultValue(Class<?> type) {
  return DEFAULT_VALUES.get(type.descriptorString());
}
```

We don't store `null` for void or any object classes because `Map.get()` returns `null` if there is no corresponding key.

# Reflection

The reflection API, the package `java.lang.reflect` , provide the capabilities to

- ask for the class of any objects at runtime
- get a description of the class using Class.getFields(), Class.getMethods() or Class.getContructors()
- get and set `Field` value, invoke `Method` or create a new instance from a `Constructor`
- get annotations and compiler information stored in the classfile (the .class file)

## Class.getMethod(), Class.getMethods(), Class.getConstructors()

To get a specific method (static or not), the method Class.getMethod() is called on a class and takes the name and the parameter types of the method (not the return type).

```
Method method;
try {
  method = String.class.getMethod("concat", String.class);
} catch(NoSuchMethodException e) {
  throw (NoSuchMethodError) new NoSuchMethodError().initCause(e);
}
```

The method may not exist, it that case the usual solution is to stop the execution by propagating an `Error` , here a `NoSuchMethodError` with the initial exception chained.

The method Class.getMethods() returns **all** the **public** methods

```
Method[] methods = String.class.getMethods();
```

## Method.invoke(), Constructor.newInstance()

The method Method.invoke calls the method with an instance and the arguments.

To be able to `invoke` a method, the method has to be accessible which means, if the declaring class is a different package, that the class that calls `invoke` that

- the method has to be public
- the declaring class of the method has to be public
- the declaring package has to be exported or open

```
void invokeMethod(Method setter, Object instance, Object[] args) {
  try {
    return method.invoke(instance, value);
  } catch (IllegalArgumentException e) {
    throw new AssertionError(e);
  } catch (IllegalAccessException e) {
    throw (IllegalAccessError) new IllegalAccessError().initCause(e);
  } catch (InvocationTargetException e) {
    throw rethrow(e.getCause());
  }
}

@SuppressWarnings("unchecked")   // very wrong but works
static <T extends Throwable> AssertionError rethrow(Throwable cause) throws T {
  throw (T) cause;
}
```

Here managing the possible exceptions is a little complex

- if the instance is not a subclass of the declaring class, `IllegalArgumentException` is raised, this should not append we propagate an AssertionError
- if the method is not accessible `IllegalAccessException` is raised, like with `NoSuchMethodException` previously, we throw the corresponding error, here `IllegalAccessError` .
- if when the method is called, the method itself raises an exception, it will be stored inside an InvocationTargetException, and we just rethrow it

Rethrowing an exception is Java is unnecessary hard because in Java the compiler want to know if an exception is checked or not. There is a hack that relies on the fact that the checked exception are only checked by the compiler and that the erasure of generics allows to see a `throws Throwable` as a `throws RuntimeException` at runtime. This is exactly what `rethrow()` does here.

The method Constructor.newInstance() calls a constructor with arguments. Like with `Method.invoke` , if a constructor is a different package, it is only accessible if the constructor is public, the declaring class is public and the declaring package is exported or open.

```
Object newInstance(Constructor<?> constructor, Object... args) {
  try {
    return constructor.newInstance(args);
  }  catch (IllegalArgumentException e) {
    throw new AssertionError(e);
  } catch (InstantiationException e) {
    throw (InstantiationError) new InstantiationError().initCause(e);
  } catch (IllegalAccessException e) {
    throw (IllegalAccessError) new IllegalAccessError().initCause(e);
  } catch (InvocationTargetException e) {
    throw rethrow(e.getCause());
```

```
    }
  }
```

Compared to `Method.invoke` , there is a supplementary exception, `InstantiationException` that is raised if the declaring class is not a concrete class. Like with `IllegalAccessException` , we propagate the corresponding error `InstantiationError` .

# Java Bean and BeanInfo

Java Bean is a **convention** to interact at runtime with a class instance sees as container of properties. The class Introspector returns a BeanInfo describing the class taken as parameter as a Java Bean.

A property is a virtual entity defined by the presence of a method that starts with `get` or `is` and/or a method that starts with `set` . By example, the code below defines a property `name` of type `String` .

```
public class Person {
  String getName() { ... }
  void setName(String name) { ... }
}
```

The method beanInfo.getPropertyDescriptors() returns the properties i.e. a name, a type and a read method also called a getter and a write method also called a setter.

```
void method(Class<?> beanType) {
  BeanInfo beanInfo;
  try {
    beanInfo = Introspector.getBeanInfo(beanType);
  } catch (IntrospectionException e) {
    throw new IllegalStateException(e);
  }

  PropertyDescriptor[] properties = beanInfo.getPropertyDescriptors();
  for(PropertyDescriptor property: properties) {
    String name = property.getName();
    Class<?> type = property.getPropertyType();
    Method getter = property.getReadMethod();
    Method setter = property.getWriteMethod();
    ...
  }
}
```

The method Introspector.decapitalize() provides the property name from a method name without its prefix `is` , `get` or `set` .

```
void method(String methodName)
  if (methodName.length() > 3 && methodName.startsWith("get")) {
    String propertyName = Introspector.decapitalize(methodName.substring(3));
    ...
  }
}
```

# Record

A record is a named tuple composed of ordered components that can be queried at runtime.

```
public record Person(String name, int age) {}
```

The method Class.isRecord() returns `true` is a class is a record. The method Class.getRecordComponents() returns an array of the record components or `null`. A record component is kind a like a Bean property, it is defined by

- a name
- a type
- an accessor (a getter, records are not mutable)

```
void method(Class<?> type) {
  if (!type.isRecord()) {
    ...
  }
  RecordComponent[] components = type.getRecordComponents();
  for(RecordComponent component: components) {
    String name = component.getName();
    Class<?> type = component.getType();
    Method accessor = component.getAccessor();
    ...
  }
}
```

# Dynamic Proxy

A *dynamic* proxy is a class generated at runtime that implement a list of interfaces, this is done by generating the classfile in an array of bytes at runtime and uses a ClassLoader to load that array of bytes as an actual class.

## InvocationHandler, Proxy.newProxyInstance()

A *dynamic* proxy uses an InvocationHandler, a functional interface that will be called to implement each method of the proxy. An `InvocationHandler` takes as parameter the proxy instance, the method to implement, and the arguments of the method call and asks for the return value.

```
InvocationHandler invocationHandler =
    (Object proxy, Method method, Object[] args) -> {
      ...
    };
Object proxy = Proxy.newProxyInstance(type.getClassLoader(),
    new Class<?>[] { type },
    invocationHandler);
```

The method Proxy.newProxyInstance takes a classloader (usually the classloader using to load the interfaces to implement), an array of interfaces to implement and an instance of `InvocationHandler` and returns an object (the proxy) that implements all the interfaces.

Each time a method is called on that proxy instance, the corresponding `InvocationHandler` is called with the interface method called.

### Proxy.isProxy() and Proxy.getInvocationHandler()

Sometimes it's important to know if an object is a proxy that is using a specific class as `InvocationHandler`. The method Proxy.isProxyClass() returns `true` is the class is a proxy class generated by a call to `Proxy.newProxyInstance`.

If an instance is a proxy instance, the method Proxy.getInvocationHandler() retrieve the `InvocationHandler` of a proxy instance so can be used to know if the proxy is a known proxy or not.

```
import java.lang.reflect.Proxy;

class MyInvocationHandler implements InvocationHandler {
  @Override
  public Object invoke(Object proxy, Method method, Object[] args) {
    ...
  }
}

void method(Object o) {
  if (Proxy.isProxyClass(o.getClass())) {
    ...
  }
  InvocationHandler invocationHandler = Proxy.getInvocationHandler(o);
  if (invocationHandler instanceof MyInvocationHandler myInvocationHandler) {
    ...
  }
}
```

# Annotation

An annotation is a user defined modifier containing a `Map` of constants values that can be queried to retrieve those values associated with a class, a field, a method, a record component, etc.

### Declaration and meta-annotations

An annotation is declared with the keyword `@interface`. At runtime, an annotation is equivalent to an interface that provides a typesafe access to a Map that stores constant values.

The method of an annotation are abstract with no parameter and only accept types that are type of constants for the compiler (primitive types, String, Enum, annotations, arrays of those types).

```
@interface MyAnnotation {
  String value();
}
```

The annotation above declare that instance of that annotation will store a value of type String.

Annotation uses meta-annotation, annotations that you declare on an annotation to specify

- if the annotation is available at runtime

- the type of elements of the language (an AnnotatedElement, i.e. classes, field methods, etc) that accept that annotation

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.CONSTRUCTOR})
@interface Marker {
}
```

If an annotation has the `Retention` `RUNTIME`, it's values will be available at runtime.

### Method.isAnnotationPresent(), Method.getAnnotation(), Method.getAnnotations()

The method AnnotatedElement.isAnnotationPresent() returns `true` if the annotated element is annotated with the annotation taken as argument.

```
void method(Class<?> type) {
  boolean isPresent = type.isAnnotationPresent(Marker.class);
  ...
}
```

The method AnnotatedElement.getAnnotation() return the annotation or `null` if the annotation is not present (or is not a runtime annotation). Once the annotation is retrieved, the constants values can be read using the methods of the annotation.

```
void method(Method method) {
  MyAnnotation myAnnotation = method.getAnnotation(MyAnnitation.class);
  if (myAnnotation != null) {
    String value = myAnnotation.value();
    ...
  }
}
```

It is also possible to get all the annotations declared on an `AnnotatedElement` using the method AnnotatedElement.getAnnotations().

```
void method(Constructor<?> constructor) {
  Annotation[] annotations = constructor.getAnnotations();
  ...
}
```

# Java Compiler generics attributes

Parametrized types, named generics in Java, are verified at compile time but disappear at runtime (*erasure*). In order to allow separate compilation ; a library, and an application compiled independently ; the compiler insert a special attribute into the classfile.

These attributes can be retrieved at runtime using the methods following the pattern `getGeneric*()` on the class, method, field, etc. For example, the method Class.getGenericInterfaces() returns the interfaces of a class as an array of Type.

A `java.lang.reflect.Type` can be

- a parameterized type
- a generic array
- a type variable
- a wildcard

A `java.lang.Class` represents a class or interface at runtime so a `Class` like `List` does not have a type argument ( `List` not `List<String>` ). A `Type` represents a type computed by the compiler, so it can be a parameterized type, with wildcards, etc.

```
void method(Class<?> type) {
  Type[]  genericInterfaces = type.getGenericInterfaces();
  for(Type genericInterface: genericInterfaces) {
    switch(genericInterface) {
      case Class<?> clazz -> ...                        // e.e. String.class
      case ParameterizedType parameterizedType -> {   // e.g. List<String>
        Class<?> rawType = (Class<?>) parameterizedType.getRawType();
        Type[] typeArguments = parameterizedType.getActualTypeArguments();
        ...
      }
      case GenericArrayType genericArrayType -> ...  // e.g. List<String>[]
      case TypeVariable<?> typeVariable -> ...       // e.g T
      case WildcardType wildcardType -> ...          // e.g ? extends String
    }
  }
}
```