MATHEMA

# Bytecodegefrickel

## Java-Klassendateien lesen, generieren und transformieren

**Michael Wiedeking · Java Forum Stuttgart · 31. Juli 2024**

- Einführung in das Format einer Klassendatei
- Einführung in den Bytecode
- Einführung in die Möglichkeiten der Class File API

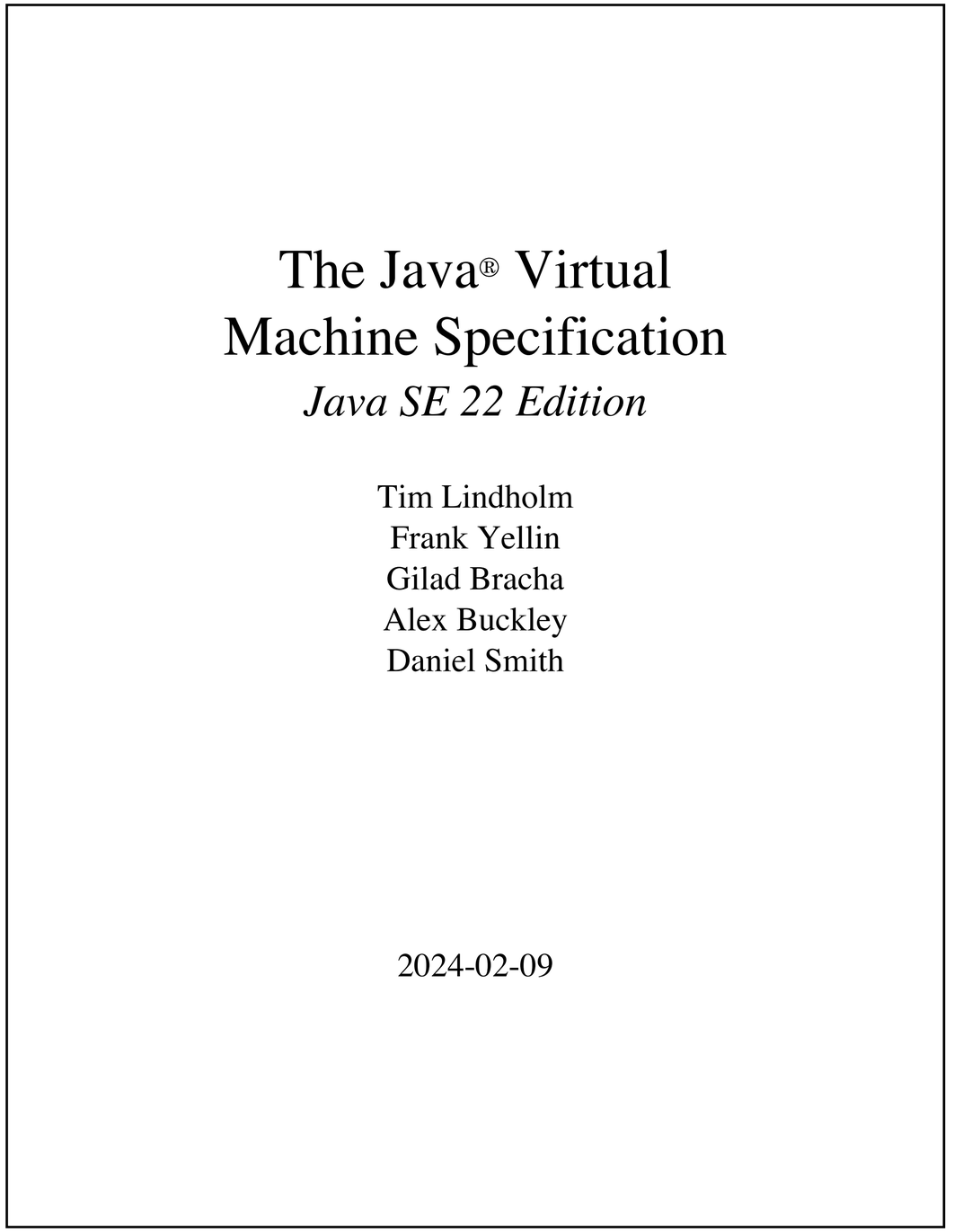
- Eine sehr kurze Einführung in das Format einer Klassendatei
- Eine sehr kurze Einführung in den Bytecode
- Eine sehr kurze Einführung in die Möglichkeiten des Class File API

We cross each bridge
when we get there...

# Teil I

```
package com.example;

import java.lang.Comparable;

public final class Example extends Object implements Comparable<Example> {

    static final int value = 1234;

    public static void main(String[] args) {

        System.out.println("Hello, World!")

        System.out.println(value)

    }

}
```

# Lieblingslektüre



# ClassFile

```
U4             magic;
U2             minor_version;
U2             major_version;
```

```
U2             cp_count;
CpInfo         constant_pool[cp_count – 1];
```

```
U2             access_flags;
C2             this_class;
C2             super_class;
```

```
U2             interfaces_count;
C2             interfaces[interfaces_count];
```

```
U2             fields_count;
FieldInfo      fields[fields_count];
U2             methods_count;
MethodInfo     methods[methods_count];
```

```
U2             attributes_count;
AttributeInfo  attributes[attributes_count];
```

# ClassFile

```
U4             magic;

U2             minor_version;

U2             major_version;
```

```
U2             cp_count;

CpInfo         constant_pool[cp_count – 1];
```

```
U2             access_flags;

C2             this_class;

C2             super_class;
```

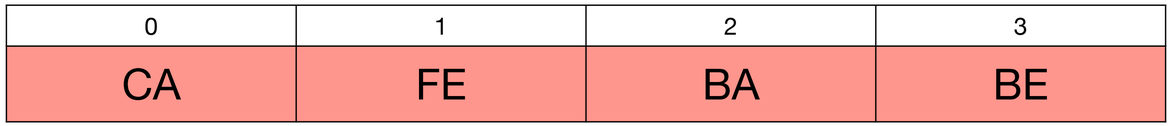
```
U2             interfaces_count;

C2             interfaces[interfaces_count];
```

```
U2             fields_count;

FieldInfo      fields[fields_count];

U2             methods_count;

MethodInfo     methods[methods_count];
```

```
U2             attributes_count;

AttributeInfo  attributes[attributes_count];
```

# ClassFile.*magic*

# 0xCafeBabe



# ClassFile.*major_version*

| Java SE | Freigabe | Major | Unterstützung |
|---|---|---|---|
| **1.0.2** | Mai 1996 | 45 | 45 |
| **1.1** | Februar 1997 | 45 | 45 |
| **1.2** | Dezember 1998 | 46 | 45, 46 |
| **1.3** | Mai 2000 | 47 | 45, ..., 47 |
| **1.4** | Februar 2002 | 48 | 45, ..., 48 |
| **5.0** | September 2004 | 49 | 45, ..., 49 |
| **6** | Dezember 2006 | 50 | 45, ..., 50 |
| **7** | Juli 2011 | 51 | 45, ..., 51 |
| **8** | März 2014 | 52 | 45, ..., 52 |
| **9** | September 2017 | 53 | 45, ..., 53 |
| **10** | März 2018 | 54 | 45, ..., 54 |
| **11** | September 2018 | 55 | 45, ..., 55 |

| Java SE | Freigabe | Major | Unterstützung |
|---|---|---|---|
| **12** | März 2019 | 56 | 45, ..., 56 |
| **13** | September 2019 | 57 | 45, ..., 57 |
| **14** | März 2020 | 58 | 45, ..., 58 |
| **15** | September 2020 | 59 | 45, ..., 59 |
| **16** | März 2021 | 60 | 45, ..., 60 |
| **17** | September 2021 | 61 | 45, ..., 61 |
| **18** | März 2022 | 62 | 45, ..., 62 |
| **19** | September 2022 | 63 | 45, ..., 63 |
| **20** | März 2023 | 64 | 45, ..., 64 |
| **21** | September 2023 | 65 | 45, ..., 65 |
| **22** | März 2024 | 66 | 45, ..., 66 |
| **23** | September 2024 | 67 | 45, ..., 67 |

# ClassFile.*minor_version*

- *major_version* ∈ {45, ..., 55}

  *minor_version*: beliebig

- *major_version* ∈ {56, ..., 67}

  *minor_version*: 0 oder 65535

| Java SE | Freigabe | Major | Unterstützung |
|---|---|---|---|
| **12** | März 2019 | 56 | 45, ..., 56 |
| **13** | September 2019 | 57 | 45, ..., 57 |
| **14** | März 2020 | 58 | 45, ..., 58 |
| **15** | September 2020 | 59 | 45, ..., 59 |
| **16** | März 2021 | 60 | 45, ..., 60 |
| **17** | September 2021 | 61 | 45, ..., 61 |
| **18** | März 2022 | 62 | 45, ..., 62 |
| **19** | September 2022 | 63 | 45, ..., 63 |
| **20** | März 2023 | 64 | 45, ..., 64 |
| **21** | September 2023 | 65 | 45, ..., 65 |
| **22** | März 2024 | 66 | 45, ..., 66 |
| **23** | September 2024 | 67 | 45, ..., 67 |

```
public static byte[] generateClass() {
    ClassFile.of class_file = ClassFile.of();
    ClassDesc class_name = ClassDesc.of("com.example", "Example");
    return class_file.build(class_name, class_builder -> { … });
}
```

```
public static byte[] generateClass() {
    ClassFile class_file = ClassFile.of();
    ClassDesc class_name = ClassDesc.of("com.example", "Example");
    return class_file.build(class_name, class_builder -> {
        class_builder
            .withVersion(65, 0)
        ;
    });
}
```

# ClassFile

```
U4             magic;

U2             minor_version;

U2             major_version;

U2             cp_count;

CpInfo         constant_pool[cp_count – 1];

U2             access_flags;

C2             this_class;

C2             super_class;

U2             interfaces_count;

C2             interfaces[interfaces_count];

U2             fields_count;

FieldInfo      fields[fields_count];

U2             methods_count;

MethodInfo     methods[methods_count];

U2             attributes_count;

AttributeInfo  attributes[attributes_count];
```

# CpInfo

U1 *tag*;

U1[ ] *info*;

# CpInfo.*tag*

- CONSTANT_Utf8
- CONSTANT_NameAndType
- CONSTANT_Class
- CONSTANT_Fieldref
- CONSTANT_Methodref
- CONSTANT_InterfaceMethodref

# CpInfo.*tag*

- CONSTANT_String
- CONSTANT_Integer
- CONSTANT_Float
- CONSTANT_Long
- CONSTANT_Double

# CpInfo.*tag*

- CONSTANT_MethodHandle
- CONSTANT_MethodType
- CONSTANT_Dynamic
- CONSTANT_InvokeDynamic
- CONSTANT_Module
- CONSTANT_Package

# CpInfo.*tag* = CONSTANT_Utf8

U1 *tag*;

U2 *length*;

U1 *bytes*[*length*];

- Wie UTF-8, aber:
- Kein Byte hat den Wert 0
- Surrogate werden einzeln in einer 3-Byte-Variante kodiert

# CpInfo.*tag* = CONSTANT_Class_info

U1 *tag*;

C2 *name_index*;

- Gültiger Index in den Constant Pool
- Muss auf ein CONSTANT_Utf8 verweisen

# CpInfo.*tag* = CONSTANT_*ref_info

U1 *tag*;

C2 *name_index*;

C2 *name_and_type_index*;

- CONSTANT_Fieldref_info
- CONSTANT_Methodref_info
- CONSTANT_InterfaceMethodref_info

# CpInfo.*tag* = CONSTANT_NameAndType

U1 *tag*;

C2 *name_index*;

C2 *descriptor_index*;

- Unqualifizierter Name
- Feld- oder Methoden-Deskriptor

# ClassFile

```
U4             magic;

U2             minor_version;

U2             major_version;
```

```
U2             cp_count;

CpInfo         constant_pool[cp_count – 1];
```

```
U2             access_flags;

C2             this_class;

C2             super_class;
```

```
U2             interfaces_count;

C2             interfaces[interfaces_count];
```

```
U2             fields_count;

FieldInfo      fields[fields_count];

U2             methods_count;

MethodInfo     methods[methods_count];
```

```
U2             attributes_count;

AttributeInfo  attributes[attributes_count];
```

# ClassFile.*access_flags*

| Name | Bedeutung |
|---|---|
| **ACC_PUBLIC** | Mit *public* deklariert. |
| **ACC_FINAL** | Mit *final* deklariert. |
| **ACC_SUPER** | Besondere Behandlung der *invokespecial* Instruktion. |
| **ACC_INTERFACE** | Schnittstelle (keine Klasse). |
| **ACC_ABSTRACT** | Mit *abstract* deklariert. |
| **ACC_SYNTHETIC** | Ohne Quellcode (vom Compiler generiert). |
| **ACC_ANNOTATION** | Schnittstelle ist Annotation. |
| **ACC_ENUM** | Aufzählung. |
| **ACC_MODULE** | Modul (weger Klasse noch Schnittstelle). |

Warum gibt es kein ACC_RECORD?

```
private static final ClassDesc CD_Comparable = ClassDesc.of("java.lang.Comparable")

public static byte[] generateClass() {
    ClassFile.of class_file = ClassFile.of();
    ClassDesc class_name = ClassDesc.of("com.example", "Example");
    return class_file.build(class_name, class_builder -> {
        class_builder
            .withFlags(ClassFile.ACC_PUBLIC + ClassFile.ACC_FINAL)
            .withSuperclass(ConstantDescs.CD_Object)
            .withInterfaceSymbols(CD_Comparable)
        ;
    });

}
```

```
private static final ClassDesc CD_Comparable = ClassDesc.of("java.lang", "Comparable")

public static byte[] generateClass() {
    ClassFile.of class_file = ClassFile.of();
    ClassDesc class_name = ClassDesc.of("com.example", "Example");
    return class_file.build(class_name, class_builder -> {
        class_builder
            .withFlags(ClassFile.ACC_PUBLIC + ClassFile.ACC_FINAL)
            .withSuperclass(ConstantDescs.CD_Object)
            .withInterfaceSymbols(CD_Comparable)
        ;
    });
}
```

```
private static final ClassDesc CD_Comparable = ClassDesc.ofDescriptor("Ljava/lang/Comparable;")

public static byte[] generateClass() {
    ClassFile.of class_file = ClassFile.of();
    ClassDesc class_name = ClassDesc.of("com.example", "Example");
    return class_file.build(class_name, class_builder -> {
        class_builder
            .withFlags(ClassFile.ACC_PUBLIC + ClassFile.ACC_FINAL)
            .withSuperclass(ConstantDescs.CD_Object)
            .withInterfaceSymbols(CD_Comparable)
        ;
    });
}
```

# ClassFile

```
U4             magic;

U2             minor_version;

U2             major_version;
```

```
U2             cp_count;

CpInfo         constant_pool[cp_count – 1];
```

```
U2             access_flags;

C2             this_class;

C2             super_class;
```

```
U2             interfaces_count;

C2             interfaces[interfaces_count];
```

```
U2             fields_count;

FieldInfo      fields[fields_count];

U2             methods_count;

MethodInfo     methods[methods_count];
```

```
U2             attributes_count;

AttributeInfo  attributes[attributes_count];
```

# FieldInfo

| | |
|---|---|
| U2 | *access_flags*; |
| C2 | *name_index*; |
| C2 | *descriptor_index*; |

| | |
|---|---|
| U2 | *attributes_count*; |
| AttributeInfo | *attributes*[*attributes_count*]; |

# FieldInfo.*access_flags*

| Name | Bedeutung |
|---|---|
| **ACC_PUBLIC** | Mit *public* deklariert. |
| **ACC_PRIVATE** | Mit *private* deklariert. |
| **ACC_PROTECTED** | Mit *protected* deklariert. |
| **ACC_STATIC** | Mit *static* deklariert. |
| **ACC_FINAL** | Mit *final* deklariert. |
| **ACC_VOLATILE** | Mit *volatile* deklariert. |
| **ACC_TRANSIENT** | Mit *transient* deklariert. |
| **ACC_SYNTHETIC** | Mit *final* deklariert. |
| **ACC_ENUM** | Aufzählungselement. |

# Field Descriptors

| Tag | Typ |
|---|---|
| **B** | byte |
| **S** | short |
| **I** | int |
| **J** | long |
| **F** | float |
| **D** | double |
| **Z** | boolean |
| **C** | char |
| **L***full_qualified_name*; | reference |
| **[** | array reference |

int *i*;

**I**

double[ ] *d*;

**[D**

Thread *t*;

java.lang.Thread *t*;

**Ljava/lang/Thread;**

```
byte[ ] bytes = class_file.build(class_name, class_builder -> {
    class_builder
        // int value;
        .withField("value", ConstantDescs.CD_int, field_builder -> {
            field_builder
                .withFlags(ClassFile.ACC_STATIC + ClassFile.ACC_FINAL)
            ;
        })
    ;
});
```

```
byte[ ] bytes = class_file.build(class_name, class_builder -> {
    class_builder
        // int value = 1234;
        .withField("value", ConstantDescs.CD_int, field_builder -> {
            field_builder
                .withFlags(ClassFile.ACC_STATIC + ClassFile.ACC_FINAL)
                .with(ConstantValueAttribute.of(Integer.valueOf(1234)))
            ;
        })
    ;
});
```

# MethodInfo

| | |
|---|---|
| U2 | *access_flags*; |
| C2 | *name_index*; |
| C2 | *descriptor_index*; |

| | |
|---|---|
| U2 | *attributes_count*; |
| AttributeInfo | *attributes*[*attributes_count*]; |

# MethodInfo.*access_flags*

| Name | Bedeutung |
|---|---|
| **ACC_PUBLIC** | Mit *public* deklariert. |
| **ACC_PRIVATE** | Mit *private* deklariert. |
| **ACC_PROTECTED** | Mit *protected* deklariert. |
| **ACC_STATIC** | Mit *static* deklariert. |
| **ACC_FINAL** | Mit *final* deklariert. |
| **ACC_SYNCHRONIZED** | Mit *synchronized* deklariert. |
| **ACC_BRIDGE** | Eine vom Compiler generierte Brückenmethode. |
| **ACC_VARARGS** | Variable Anzahl von Argumenten. |
| **ACC_NATIVE** | Mit *native* deklariert. |
| **ACC_ABSTRACT** | Mit *abstract* deklariert. |
| **ACC_STRICT** | Mit *strict* deklariert. |
| **ACC_SYNTHETIC** | Mit *final* deklariert. |

# Method Descriptors

| Tag | Typ |
|---|---|
| **B** | byte |
| **S** | short |
| **I** | int |
| **J** | long |
| **F** | float |
| **D** | double |
| **Z** | boolean |
| **C** | char |
| **L***full_qualified_name*; | reference |
| **[** | array reference |
| **V** | void |

Object exampleMethod(int *i*, double *d*, Thread *t*)

java.lang.Object exampleMethod(int *i*, double *d*, java.lang.Thread *t*)

**(IDLjava/lang/Thread;)Ljava/lang/Object;**

```
MethodDesc.of(
    ConstantDescs.CD_Object,
    ConstantDescs.CD_int,
    ConstantDescs.CD_double,
    ClassDesc.of("java.lang.Thread")
)
```

Object exampleMethod(int *i*, double[] *d*, Thread *t*)

java.lang.Object exampleMethod(int *i*, double[] *d*, java.lang.Thread *t*)

**(I[DLjava/lang/Thread;)Ljava/lang/Object;**

```
MethodDesc.of(
    ConstantDescs.CD_Object,
    ConstantDescs.CD_int,
    ConstantDescs.CD_double.arrayType(),
    ClassDesc.of("java.lang.Thread")
)
```

void main(String[] *args*)

void main(java.lang.String[] *args*)

**([Ljava/lang/String;)V**

```
MethodTypeDesc.of(
    ConstantDescs.CD_void,
    ConstantDescs.CD_String.arrayType()
);
```

# ClassFile

```
U4             magic;

U2             minor_version;

U2             major_version;
```

```
U2             cp_count;

CpInfo         constant_pool[cp_count – 1];
```

```
U2             access_flags;

C2             this_class;

C2             super_class;
```

```
U2             interfaces_count;

C2             interfaces[interfaces_count];
```

```
U2             fields_count;

FieldInfo      fields[fields_count];

U2             methods_count;

MethodInfo     methods[methods_count];
```

```
U2             attributes_count;

AttributeInfo  attributes[attributes_count];
```

# *Member*Info

| | |
|---|---|
| U2 | *access_flags*; |
| C2 | *name_index*; |
| C2 | *descriptor_index*; |
| U2 | *attributes_count*; |
| AttributeInfo | *attributes*[*attributes_count*]; |

# AttributeInfo

C2 *attribute_name_index*;

U4 *attribute_length*;

U1 *attributes*[*attributes_length*];

# Deprecated Attribute

| | | |
|---|---|---|
| C2 | *attribute_name_index*; | // “Deprecated” |
| U4 | *attribute_length*; | // 0 |

# SourceFile Attribute

```
C2      attribute_name_index;      // “SourceFile”
U4      attribute_length;          // 2
C2      source_file_index;         // Verweis auf den Dateinamen
```

```
byte[ ] bytes = class_file.build(class_name, class_builder -> {
    class_builder
        // int value = 123;
        .withField("value", ConstantDescs.CD_int, field_builder -> {
            field_builder
                .withFlags(ClassFile.ACC_STATIC + ClassFile.ACC_FINAL)
                .with(ConstantValueAttribute.of(Integer.valueOf(123)))
            ;
        })
    ;
});
```

# Standardattribute („kritisch“)

- ConstantValue
- Code
- StackMapTable
- BootstrapMethods
- NestHost
- NestMembers
- PermittedSubclasses

# Standardattribute („optional“)

- Record
- InnerClasses
- EnclosingMethod
- Synthetic
- SourceFile
- Exceptions
- LineNumberTable
- LocalVariableTable
- LocalVariableTypeTable
- Signature

# Standardattribute („meta“)

- SourceDebugExtension
- Deprecated
- MethodParameters

- RuntimeVisibleAnnotations
- RuntimeVisibleParameterAnnotations
- RuntimeVisibleTypeAnnotations

- RuntimeInvisibleAnnotations
- RuntimeInvisibleParameterAnnotations
- RuntimeInvisibleTypeAnnotations
- AnnotationDefault

- Module
- ModulePackages
- ModuleMainClass

# Standardattribute („kritisch“)

- ConstantValue
- **Code**
- StackMapTable
- BootstrapMethods
- NestHost
- NestMembers
- PermittedSubclasses

```
// MethodTypeDesc.of(ClassDesc out_type, ClassDesc ... in_types)
private static final MethodTypeDesc MTD_void_String = MethodTypeDesc.of(
        ConstantDescs.CD_void,
        ConstantDescs.CD_String
);
private static final MethodTypeDesc MTD_void_StringArray = MethodTypeDesc.of(
        ConstantDescs.CD_void,
        ConstantDescs.CD_String.arrayType()
);
```

```
Consumer<ClassBuilder> class_builder = cb -> cb
    .withMethod(
        "main",
        MTD_void_StringArray,
        ClassFile.ACC_PUBLIC + ClassFile.ACC_STATIC,
        main_method_builder
    )
;
```

```
Consumer<MethodBuilder> main_method_builder = mb -> mb.withCode(
    code_builder -> {
        code_builder
            .getstatic(CD_System, "out", CD_PrintStream)
            .ldc("Hello, World!")
            .invokevirtual(CD_PrintStream, "println", MTD_void_String)
            .return_()
        ;
    }
);
```

```
Consumer<MethodBuilder> mainMethodBuilder(String text) {
    return mb -> mb.withCode(code_builder -> code_builder
        .getstatic(CD_System, "out", CD_PrintStream)
        .ldc(text)
        .invokevirtual(CD_PrintStream, "println", MTD_void_String)
        .return_()
    );
}
```

```
Consumer<ClassBuilder> class_builder = cb -> cb
    .withMethod(
        "main",
        MTD_void_StringArray,
        ClassFile.ACC_PUBLIC + ClassFile.ACC_STATIC,
        mainMethodBuilder("Hello world!")
    )
;
```

```
ClassDesc class_name = ClassDesc.of("com.example", "Example");

Consumer<MethodBuilder> main_method_builder = mb -> mb.withCode(

    code_builder -> code_builder

        .getstatic(CD_System, "out", CD_PrintStream)

        .getstatic(class_name, "value", ConstantDescs.CD_int)

        .invokevirtual(CD_PrintStream, "println", MTD_void_int)

        .return_()

    ;

);
```

```
.withMethod(
    ConstantDescs.INIT_NAME,                    // <init>
    ConstantDescs.MTD_void,                     // ()V
    ClassFile.ACC_PUBLIC,
    mb -> mb.withCode(cob -> cob
        .aload(0)                               // this
        .invokespecial(                         // java.lang.Object <init> ()V
            ConstantDescs.CD_Object, ConstantDescs.INIT_NAME, ConstantDescs.MTD_void
        )
        .return_()
    )
)
```

# Teil II

```
package com.example;

import java.lang.Comparable;

public final class Example extends Object implements Comparable<Example> {

    static final int value = 1234;

    public static void main(String[] args) {

        System.out.println("Hello, World!")

        System.out.println(value)

    }

}
```

```
> javap com.example.Example
```

```
public final class com.example.Example implements java.lang.Comparable {
    public static final int value;
    public static void main(java.lang.String[]);
}
```

```
> javap -s com.example.Example
```

```
public final class com.example.Example implements java.lang.Comparable {
    public static final int value;
        descriptor: I
    public static void main(java.lang.String[]);
        descriptor: ([Ljava/lang/String;)V
}
```

```
static void inspectClass(byte[ ] class_bytes) {
    ClassFile class_file = ClassFile.of();
     ClassModel class_model = class_file.parse(class_bytes);
     System.out.println(class_model.thisClass().name());
     for (ClassElement element : class_model) {
         System.out.println("\t" + element);
     }
}
```

com/example/Example

AccessFlags[flags=17]

ClassFileVersion[majorVersion=66, minorVersion=0]

Superclass[superclassEntry=java/lang/Object]

Interfaces[interfaces=java/lang/Comparable]

FieldModel[fieldName=value, fieldType=I, flags=25]

MethodModel[methodName=<init>, methodType=()V, flags=1]

MethodModel[methodName=main, methodType=([Ljava/lang/String;)V, flags=9]

```
package com.example;

import java.lang.Comparable;

public final class Example extends Object implements Comparable<Example> {

    static final int value = 1234;

    public static void main(String[] args) {

        System.out.println("Hello, World!")

        System.out.println(value)

    }

}
```

```
> javap -v com.example.Example
```

```
public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V
    flags: (0x0009) ACC_PUBLIC, ACC_STATIC
    Code:
        stack=2, locals=1, args_size=1
             0: getstatic     #8    // Field java/lang/System.out:Ljava/io/PrintStream;
             3: ldc           #10   // String Hello World
             5: invokevirtual #16   // Method java/io/PrintStream.println:(Ljava/lang/String;)V
             8: getstatic     #8    // Field java/lang/System.out:Ljava/io/PrintStream;
            11: getstatic     #20   // Field value:I
            14: invokevirtual #23   // Method java/io/PrintStream.println:(I)V
            17: return
    StackMapTable: number_of_entries = 1
        frame_type = 20 /* same */
```

```
static byte[ ] copyClass(byte[ ] bytes) {
    var class_file = ClassFile.of();
    var class_model = class_file.parse(bytes);
    var original = class_model.thisClass().asSymbol();
    var new_name = ClassDesc.of(original.packageName(), original.displayName() + "Copy");
    return class_file.build(
        new_name,
        class_builder -> {
            for (ClassElement element : class_model) {
                class_builder.with(element);
            }
        }
    );
}
```

```
> javap -v com.example.ExampleCopy

public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V
    flags: (0x0009) ACC_PUBLIC, ACC_STATIC
    Code:
        stack=2, locals=1, args_size=1
             0: getstatic     #8        // Field java/lang/System.out:Ljava/io/PrintStream;
             3: ldc           #10       // String Hello World
             5: invokevirtual #16       // Method java/io/PrintStream.println:(Ljava/lang/String;)V
             8: getstatic     #8        // Field java/lang/System.out:Ljava/io/PrintStream;
            11: getstatic     #30       // Field com/example/Example.value:I
            14: invokevirtual #23       // Method java/io/PrintStream.println:(I)V
            17: return
    StackMapTable: number_of_entries = 1
        frame_type = 20 /* same */
```

```
> javap -v com.example.ExampleCopy

public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V
    flags: (0x0009) ACC_PUBLIC, ACC_STATIC
    Code:
        stack=2, locals=1, args_size=1
             0: getstatic     #8        // Field java/lang/System.out:Ljava/io/PrintStream;
             3: ldc           #10       // String Hello World
             5: invokevirtual #16       // Method java/io/PrintStream.println:(Ljava/lang/String;)V
             8: getstatic     #8        // Field java/lang/System.out:Ljava/io/PrintStream;
            11: getstatic     #30       // Field com/example/Example.value:I
            14: invokevirtual #23       // Method java/io/PrintStream.println:(I)V
            17: return
    StackMapTable: number_of_entries = 1
        frame_type = 20 /* same */
```

```
CodeTransform codeTransform = (code_builder, e) -> {
    switch (e) {
        default -> {
            code_builder.accept(e);
        }
    }
};
```

```
case FieldInstruction i when i.owner().asSymbol().equals(original_name) -> {
    code_builder.fieldInstruction(
        i.opcode(),
        new_class_name,
        i.name().stringValue(),
        i.typeSymbol()
    );
}
```

```
> javap -v com.example.ExampleCopy

public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V
    flags: (0x0009) ACC_PUBLIC, ACC_STATIC
    Code:
        stack=2, locals=1, args_size=1
             0: getstatic     #8        // Field java/lang/System.out:Ljava/io/PrintStream;
             3: ldc           #10       // String Hello World
             5: invokevirtual #16       // Method java/io/PrintStream.println:(Ljava/lang/String;)V
             8: getstatic     #8        // Field java/lang/System.out:Ljava/io/PrintStream;
            11: getstatic     #30       // Field value:I
            14: invokevirtual #23       // Method java/io/PrintStream.println:(I)V
            17: return
    StackMapTable: number_of_entries = 1
        frame_type = 20 /* same */
```

```
case FieldInstruction i when i.owner().asSymbol().equals(original_name) -> {
    code_builder.fieldInstruction(
        i.opcode(),
        new_class_name,
        i.name().stringValue(),
        i.typeSymbol()
    );
}
```

```
case InvokeInstruction i when i.owner().asSymbol().equals(original_name) -> {
    code_builder.invokeInstruction(
        i.opcode(),
        new_class_name,
        i.name().stringValue(),
        i.typeSymbol(),
        i.isInterface()
    );
}
```

```
private static byte[] stripClassOfDebugMethods(byte[] bytes) {
    ClassFile class_file = ClassFile.of();
    ClassModel class_model = class_file.parse(bytes);
    byte[ ] result = … ;
    return result;
}
```

```
byte[ ] result = class_file.build(
    class_model.thisClass().asSymbol(),
    class_builder -> {
        for (ClassElement e : class_model) {
            if (isMethod(e) && !isDebugMethod(e)) {
                class_builder.with(e);
            } else {
                class_builder.with(e);
            }
        }
    }
);
```

```
byte[ ] result = class_file.build(
    class_model.thisClass().asSymbol(),
    class_builder -> {
        for (ClassElement e : class_model) {
            if (isMethod(e) &&!isDebugMethod(e)) {
                transformMethod(class_builder, (MethodModel) e);
            } else {
                code_builder.with(e);
            }
        }
    }
);
```

```
static void transformMethod(ClassBuilder class_builder, MethodModel method_model) {
    class_builder.withMethod(
        method_model.methodName(),
        method_model.methodType(),
        method_model.flags().flagsMask(),
        method_builder -> {
            for (MethodElement method_element : method_model) {
                ...
            }
        }
    );
}
```

```
if (method_element instanceof CodeModel code_model) {
    … // Code transformieren.
} else {
    method_builder.with(method_element);
}
```

```
if (method_element instanceof CodeModel code_model) {
    methodBuilder.withCode(code_builder -> {
        removeDebugCalls(code_builder, code_model);
    });
} else {
    method_builder.with(method_element);
}
```

```
static void removeDebugCalls(CodeBuilder code_builder, CodeModel code_model) {
    for (CodeElement e : code_model) {
        switch (e) {
            case InvokeInstruction i when isDebugMethod(i.method()) -> {
                // Alle hierfür auf dem Stack liegenden Werte entfernen.
            }
            default -> {
                code_builder.with(e);
            }
        }
    }
}
```

```
static void removeDebugCalls(CodeBuilder code_builder, CodeModel code_model) {
    for (CodeElement e : code_model) {
        switch (e) {
            case InvokeInstruction i when isDebugMethod(i.method()) -> {
                code_builder.pop();
            }
            default -> {
                code_builder.with(e);
            }
        }
    }
}
```

```
static void removeDebugCalls(CodeBuilder code_builder, CodeModel code_model) {
    for (CodeElement e : code_model) {
        switch (e) {
            case InvokeInstruction i when isDebugMethod(i.method()) -> {
                code_builder.pop2();
            }
            default -> {
                code_builder.with(e);
            }
        }
    }
}
```

```
static void removeDebugCalls(CodeBuilder code_builder, CodeModel code_model) {
    for (CodeElement e : code_model) {
        switch (e) {
            case InvokeInstruction i when isDebugMethod(i.method()) -> {
                code_builder.pop2();
            }
            default -> {
                code_builder.with(e);
            }
        }
    }
}
```

```
static void removeDebugCalls(CodeBuilder code_builder, CodeElement e) {
    switch (e) {
        case InvokeInstruction i when isDebugMethod(i.method()) -> {
            // Alle hierfür auf dem Stack liegenden Werte entfernen.
        }
        default -> {
            code_builder.with(e);
        }
    }
}
```

```
CodeTransform codeTransform = (CodeBuilder code_builder, CodeElement e) -> {
    switch (e) {
        case InvokeInstruction i when isDebugMethod(i.method()) -> {
            // Alle hierfür auf dem Stack liegenden Werte entfernen.
        }
        default -> {
            code_builder.with(e);
        }
    }
}
```

```
CodeTransform code_transform = (CodeBuilder code_builder, CodeElement e) -> {
    switch (e) {
        case InvokeInstruction i when isDebugMethod(i.method()) -> {
            // Alle hierfür auf dem Stack liegenden Werte entfernen.
        }
        default -> {
            code_builder.with(e);
        }
    }
}
MethodTransform method_transform = MethodTransform.transformingCode(code_transform);
```

```
CodeTransform code_transform = (CodeBuilder code_builder, CodeElement e) -> {
    switch (e) {
        case InvokeInstruction i when isDebugMethod(i.method()) -> {
            // Alle hierfür auf dem Stack liegenden Werte entfernen.
        }
        default -> {
            code_builder.with(e);
        }
    }
}
MethodTransform method_transform = MethodTransform.transformingCode(code_transform);
ClassTransform class_transform = ClassTransform.transformingMethods(method_transform);
```

```
CodeTransform code_transform = (CodeBuilder code_builder, CodeElement e) -> {
    switch (e) {
        case InvokeInstruction i when isDebugMethod(i.method()) -> {
            // Alle hierfür auf dem Stack liegenden Werte entfernen.
        }
        default -> {
            code_builder.with(e);
        }
    }
}
MethodTransform method_transform = MethodTransform.transformingCode(code_transform);
ClassTransform class_transform = ClassTransform.transformingMethods(method_transform);
byte[] result = class_file.transform(class_model, new_class_name, class_transform);
```

ClassTransform:

```
    static ClassTransform transformingMethodBodies(
        Predicate<MethodModel> filter,
        CodeTransform xform
    );
```

MethodTransform:

```
    static MethodTransform dropping(
        Predicate<MethodElement> filter
    );
```


# Fragen!?


# Vielen Dank!