TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Technische Informatik, Professur für VLSI-Entwurfssysteme, Diagnostik und Architektur

Studienarbeit:

Unterstützung von Java 8-Fähigkeiten auf der SHAP-Plattform

Alfred Krohmer

Dresden, 26.11.15

Gliederung

1. **Einleitung**
2. **SHAP-Plattform**
3. **Neuerungen in Java 8**
4. **Stand der Technik**
5. **Implementierung**

Einleitung

- Java in allen Branchen und Bereichen anzutreffen
  - Industrieanlagen
  - Unterhaltungselektronik
  - Verkehrsanlagen
  - Serveranwendungen
  - Rich Client Anwendungen
  - …
- Anforderungen:
  - Energieeffizienz
  - Performance
  - Echtzeitfähigkeit

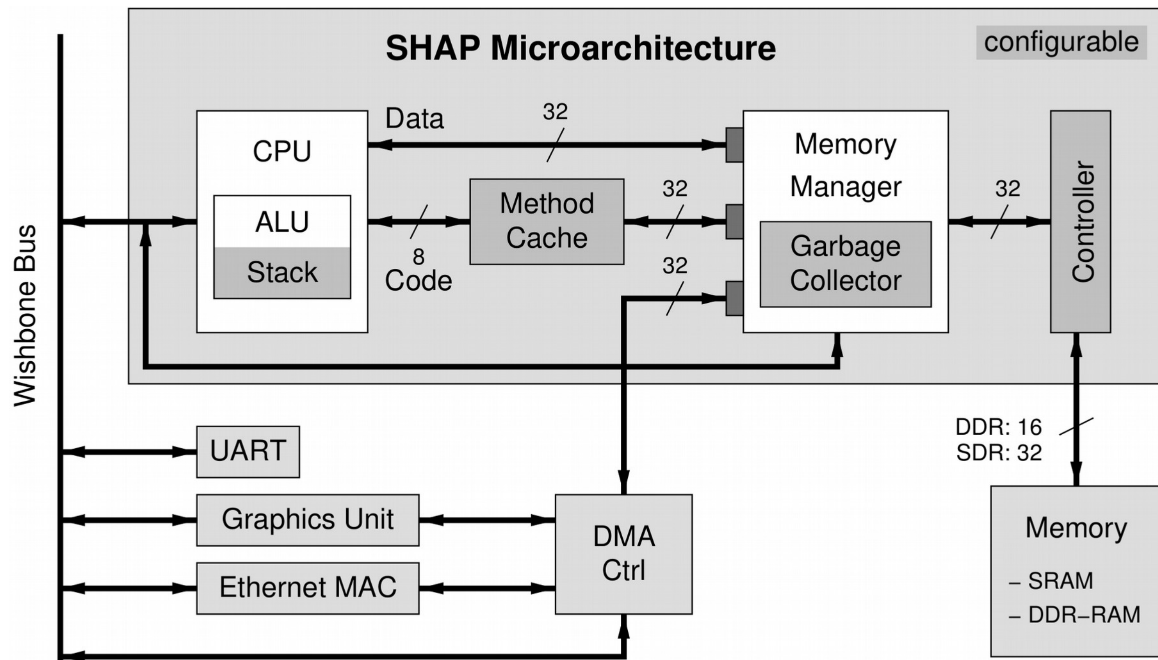
→ Realisierung als eigenständiger *Java Bytecode-Prozessor*

SHAP-Plattform

- **S**ecure **H**ardware **A**gent **P**latform
  - führt Java-Anwendungen direkt aus
  - Threads können auf mehrere Cores laufen
  - echtzeitfähig
  - Garbage Collection in Hardware
  - Dynamic Class Loading
- entstanden an der TU Dresden

TECHNISCHE UNIVERSITÄT DRESDEN

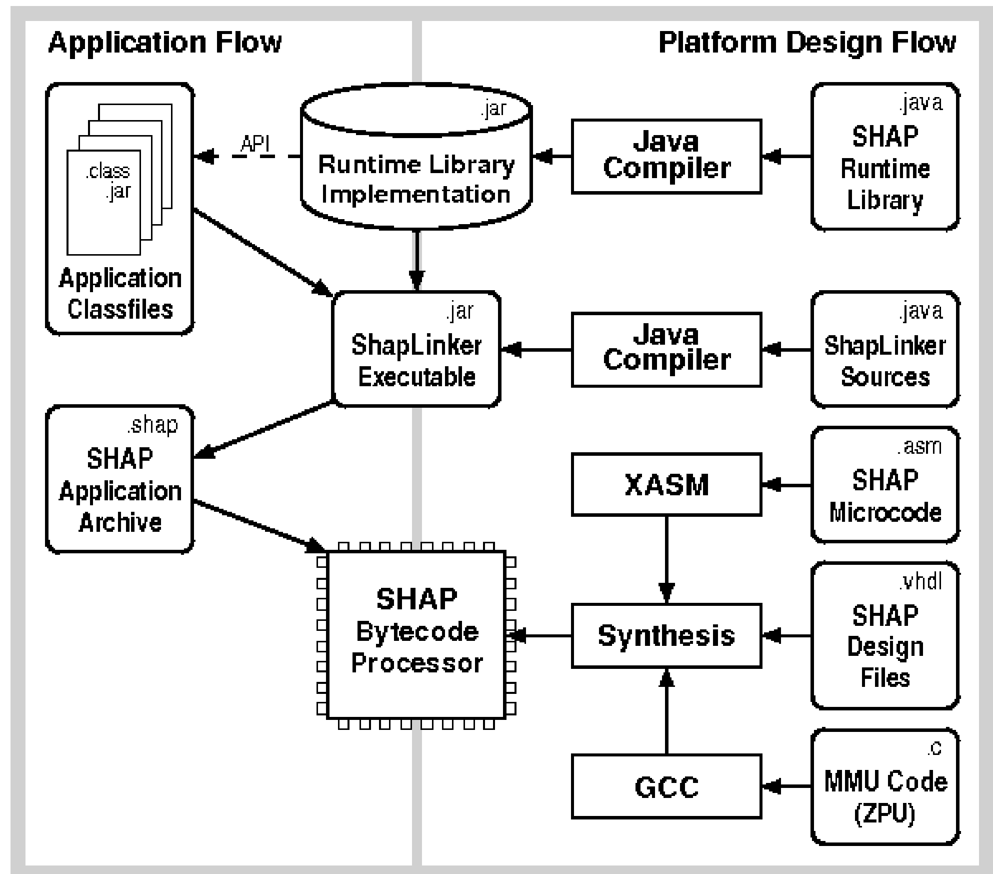
SHAP-Plattform



Quelle: SHAP Reference Manual

SHAP-Plattform

- Anwendungsentwicklung:
  - Java-Code
  - Java-Compiler
  - SHAP-Linker
  - SHAP-Prozessor



Quelle: The SHAP Bytecode Processor, http://shap.inf.tu-dresden.de

TECHNISCHE UNIVERSITÄT DRESDEN

SHAP-Plattform

- aktuell: *Java SE 7* wird unterstützt
- **Ziele der Studienarbeit**:
  - Literaturstudium zu *Java 8*-Spreachfeatures
  - Analyse zum Stand der Technik eingebetteter Java-Plattformen
  - Konzeption zur Implementierung neuer Features
  - Implementierung einiger Features im Linker
- Motivation:
  - kompakterer Code
  - funktionale Programmierung möglich
  - Kompatibilität mit existierenden Bibliotheken / fremdem Code

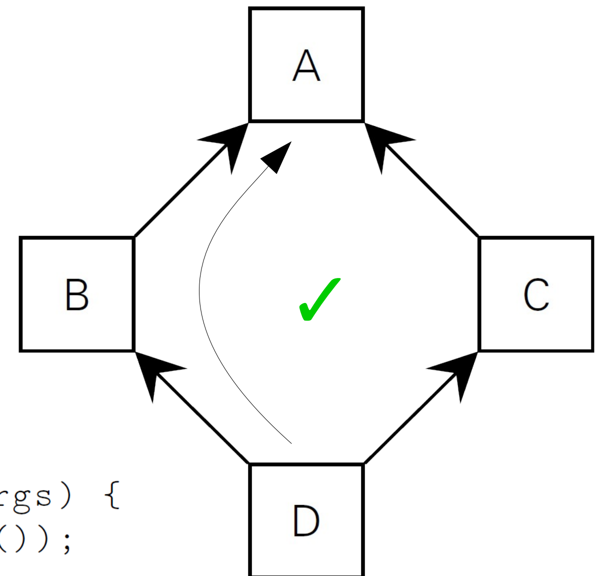
Neuerungen in Java 8: **Default Methods**

```
1 interface MyInterface {
2     default int foo() {
3         return 42;
4     }
5 }
```

Neuerungen in Java 8: **Default Methods**

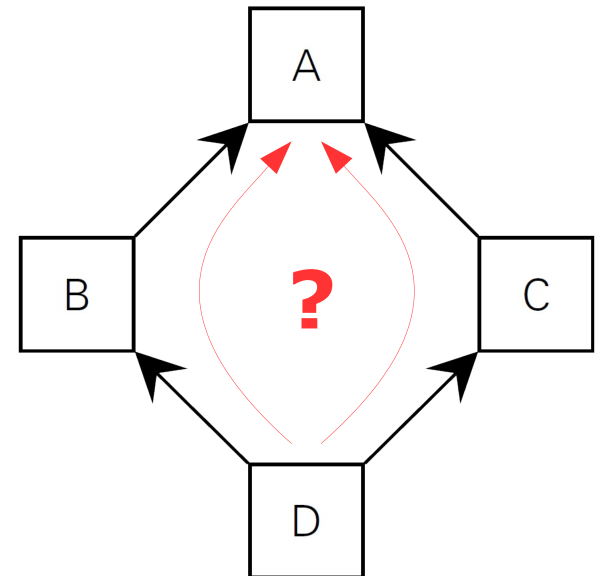
- Mehrfachvererbung → Diamond-Problem:

```
1  interface A {
2      default int foo() { return 42; }
3  }
4  interface B extends A {
5      default int foo() { return 23; }
6  }
7  interface C extends A { }
8
9  public class D implements B, C {
10     public static void main(String[] args) {
11         System.out.println(new D().foo());
12     }
13 }
```



Neuerungen in Java 8: **Default Methods**

- Mehrfachvererbung → Diamond-Problem:




```
1 // ...
2 interface C extends A {
3     default int foo() { return 1337; }
4 }
5 // ...
```

```
error: class D inherits unrelated defaults for foo() from types B and C
```

TECHNISCHE UNIVERSITÄT DRESDEN

Neuerungen in Java 8: **Functional Interface**

- Annotation: `@FunctionalInterface`
- Interface, das **genau eine** abstrakte Methode hat
- Paket: `java.util.function`:
  - Consumer: `void accept(T);`
  - BiConsumer `void accept(T, S);`
  - Supplier: `T get();`
  - Function: `R apply();`
  - Predicate: `boolean test(T);`

Neuerungen in Java 8: **Functional Interface**

- Realisierung:
  - benannte Klasse:

```
1 public class MyFunctionalClass implements Consumer<String> {
2     public void accept(String) { /* ... */ }
3 }
4 // ...
5 Consumer<String> c = new MyFunctionalClass();
```

  - anonyme Klasse:

```
1 Consumer<String> c = new Consumer<String> {
2     public void accept(String) { /* ... */ }
3 }
```

  - **Method References, Lambdas**

Neuerungen in Java 8: **Method References**

- Abbildung:
  - Referenzen auf Methoden und Konstruktoren → Functional Interface
  - Aufruf der Interface-Methode → Aufruf der referenzierten Methode (Forwarding; Proxy-Methode)

- vier Typen von Method References

Neuerungen in Java 8: **Method References**

- Referenzen auf **statische** Methoden:

```
class String {
    // …
    public static String valueOf(int i) { /* … */ }
    // …
}
// …
Function<String, Integer> f = String::valueOf;
f.apply(42); // “42“
```

Neuerungen in Java 8: **Method References**

- Referenzen auf Methoden **mit Instanzbindung**:

```
class String {
    // …
    public int length() { /* … */ }
    // …
}
// …
String str = “Hello world!“;
Supplier<Integer> s = str::length;
s.get(); // 12
```

Neuerungen in Java 8: **Method References**

- Referenzen auf Methoden **ohne Instanzbindung**:

```
class String {
    // …
    public boolean isEmpty() { /* … */ }
    // …
}
// …
Predicate<String> p = String::isEmpty;
p.test(“0“); // false
```

Neuerungen in Java 8: **Method References**

- Referenzen auf **Konstruktoren**:

```
class String {
    // …
    public String() { /* … */ }
    // …
}
// …
Supplier<String> s = String::new;
p.get(); // instance of String
```

Neuerungen in Java 8: **Lambdas**

```
1  import java.util.function.BiConsumer;
2  import java.util.function.Function;
3
4  public class Test {
5      private int a = 42;
6
7      public int doThings() {
8          int b = 23;
9
10         BiConsumer<String, String> bc = (String s1, String s2) -> {
11             System.out.println(s1 + a + s2);
12         };
13         bc.accept("The cake", "is a lie.");
14
15         Function<Integer, Integer> f = (Integer x) -> x * 5 + b;
16         return f.apply(5);
17     }
18 }
```

Stand der Technik

Version	Veröffentl.	wichtige neue Sprachfeatures	Änderung
JDK 1.1	19.02.1997	• innere Klassen	F
		• Reflection	L, B
J2SE 1.2	08.12.1998	• `strictfp`	F, B
		• Collections-Framework	L
		• Weak References und Reference Queues	L, B
J2SE 1.3	08.05.2000	-	
J2SE 1.4	06.02.2002	• `assert`	F, L
		• Exception chaining	L
J2SE 5.0	30.09.2004	• Generics	F, L
		• Annotationen	F, L
		• Aufzählungen (`enum`)	F
Java SE 6	11.12.2006	-	
Java SE 7	28.07.2011	• Strings als Keys in `switch`-Statements	F
		• verbessere Typinferenz (*Diamond-Operator* <>)	F
		• Binärzahlen als Literale	F
Java SE 8	18.03.2014	• *Default Methods* in Interfaces; damit:	F
		• (eingeschränkte) Mehrfachvererbung (B)	
		• Methodenreferenzen	F, B
		• Lambdas F, B	

Stand der Technik: **SHAP**

- CLDC 1.1
- Java SE 7
- (bisher noch) nicht unterstützt:
  - Java 8-Features
  - `assert`-Keyword

Stand der Technik: **JOP**

- CLDC 1.1
- Features auf Niveau von J2SE 5.0
- nicht unterstützt:
    - Features aus Java SE 6, 7, 8
    - `assert`-Keyword
    - Weak References
    - Reference Queues

Stand der Technik: **aJile Systems**

- direkt in Hardware implementierte Bytecode-Prozessoren

  (früher: aj-80, aj-100; heute: aj-102, aj-200)
- **CDC** 1.1
  - höherer Funktionsumfang
- letztes SDK-Update: 2010

  → Features höchstens auf Niveau von Java SE 6

Stand der Technik: **Jazelle DBX, ThumbEE**

- Erweiterungen für ARM-Prozessoren
- *Umschaltung* von Instruction Sets
- Jazelle DBX (Direct Bycode eXecution)
- Nachfolger: ThumbEE, auch Jazelle RCT (Runtime Compilation Target)
  - keine direkte Ausführung von Java-Bytecode mehr
  - Ende 2011 aufgegeben
- keine komplette Java-*Plattform*

Stand der Technik: weitere Prozessoren

- Cjib
- picoJava, picoJava II
- FemtoJava
- Komodo

Implementierung: **Default Methods**

- Überprüfung aller implementierten Interface einer Klasse:
  - alle Methoden überschrieben? → kein Fehler
  - wenn Methode nicht überschrieben → Fehler

Implementierung: **Default Methods**

- Überprüfung aller implementierten Interface einer Klasse:
  - alle Methoden überschrieben? → kein Fehler
  - wenn Methode nicht überschrieben
    → hat Interface-Methode eine *Default Implementation*?
    - Ja → kein Fehler
    - Nein → Fehler

Implementierung: **Default Methods**

- Diamond-Problem:
    - **mehrere** Implementierungen für die **gleiche** Methoden-Signatur können auftreten
    - keine Mehrfachvererbung → neue Implementierung ersetzt alte

Implementierung: **Default Methods**

- Diamond-Problem:
  - **mehrere** Implementierungen für die **gleiche** Methoden-Signatur können auftreten
  - Überprüfung: kommt die zuletzt gefundene Methode aus einer *Spezialisierung* der Klasse der zuvor gefundenen Methode?
    - Ja → kein Fehler, normale Vererbungsstrategie
    - Nein → Fehler, Diamond-Problem

Implementierung: **Functional Interfaces**

- ausgewählte Interfaces wurden in die Laufzeitbibliothek von SHAP übernommen

Implementierung: **Method References, Lambdas**

- Abbildung der Method References auf Aufruf der
  `invokedynamic`-Instruktion

- Beispiel:

```
Function<String, int> f = String::valueOf;
f.apply(42);
```

- Abbildung auf Bytecode:

```
invokedynamic    #1:apply:()Ljava/util/function/Function;
bipush           42
invokeinterface  java/util/function/Function.apply:(I)Ljava/lang/String;
```

TECHNISCHE UNIVERSITÄT DRESDEN

Implementierung: **Method References, Lambdas**

- *Bootstrap Method #1*:

```
1: invokestatic java/lang/invoke/LambdaMetafactory.metafactory:(
     Ljava/lang/invoke/MethodHandles$Lookup;
     Ljava/lang/String;
     Ljava/lang/invoke/MethodType;
     Ljava/lang/invoke/MethodType;
     Ljava/lang/invoke/MethodHandle;
     Ljava/lang/invoke/MethodType;)
   Ljava/lang/invoke/CallSite;

  Method arguments:
    (I)Ljava/lang/Object;
    invokestatic String.valueOf:(I)Ljava/lang/String;
    (I)Ljava/lang/String;
```

„Tag"

Implementierung: **Method References, Lambdas**

- `invokedynamic`-Instruktion erstellt ein *Functional Object*, das ein *Functional Interface* implementiert
- Ansatz im Linker:
  - alle `invokedynamic`-Instruktionen im Code suchen
  - Generierung einer Klasse, die das Functional Interface implementiert
  - Ersetzung der `invokedynamic`-Instruktion durch Instruktionen zum Instanzieren dieser Klasse, dabei:
  - Übergabe von *capture*-Variablen

TECHNISCHE UNIVERSITÄT DRESDEN

Implementierung: **Method References, Lambdas**

```
1  import java.util.function.BiConsumer;
2  import java.util.function.Function;
3
4  public class Test {
5      private int a = 42;
6
7      public int doThings() {
8          int b = 23;
9
10         BiConsumer<String, String> bc = (String s1, String s2) -> {
11             System.out.println(s1 + a + s2);
12         };
13         bc.accept("The cake", "is a lie.");
14
15         Function<Integer, Integer> f = (Integer x) -> x * 5 + b;
16         return f.apply(5);
17     }
18 }
```

Implementierung: **Method References, Lambdas**

- Compiler generiert:

```
private void lambda$doThings$0(String s1, String s2) {
    System.out.println(s1 + a + s2);
}

private static int lambda$doThings$1(int b, int x) {
    return x * 5 + b;
}
```

→ Behandlung wie Method References

TECHNISCHE UNIVERSITÄT DRESDEN

Implementierung: **Method References, Lambdas**

- Linker generiert:

```
public class Test$$mr$lambda$doThings$0 implements BiConsumer<String, String> {
    private Test capture0;
    public static final Test$$mr$lambda$doThings$0 new(Test i) {
        Test$$mr$lambda$doThings$0 _instance = new Test$$mr$lambda$doThings$0();
        _instance.capture0 = i;
        return _instance;
    }
    public final void accept(String s1, String s2) {
        this.capture0.lambda$doThings$0(s1, s2);
    }
}
public class Test$$mr$lambda$doThings$1 implements Function<int, int> {
    private int capture0;
    public static final Test$$mr$lambda$doThings$1 new(int b) {
        Test$$mr$lambda$doThings$1 _instance = new Test$$mr$lambda$doThings$1();
        _instance.capture0 = b;
        return _instance;
    }
    public final void accept(int x) {
        Test.lambda$doThings$1(this.capture0, x);
    }
}
```

Capture

Erzeugung Function Object

Proxy-Methode

TECHNISCHE UNIVERSITÄT DRESDEN

Implementierung: **Method References, Lambdas**

- Bytecode:

```
aload_0
invokedynamic   #1:apply:(I)Ljava/util/function/BiConsumer;
ldc             "The cake"
ldc             "is a lie."
invokeinterface java/util/function/BiConsumer.accept:(Ljava/lang/String;
                                                      Ljava/lang/String;)V
```

- transformiert zu:

```
aload_0
invokestatic    Test$$mr$lambda$doThings$0.new()LTest$$mr$lambda$doThings$0;
ldc             "The cake"
ldc             "is a lie."
invokeinterface java/util/function/BiConsumer.accept:(Ljava/lang/String;
                                                      Ljava/lang/String;)
```

Implementierung: **assert-Keyword**

- Implementierung der Klasse `java.lang.AssertionError`
- Implementierung der Methode `Class.desiredAssertionStatus()`

Konzeption: **weitere Funktionalität**

- `invokedynamic`: ursprünglich gedacht, um dynamisch typisierte Sprachen auf der JVM ausführen zu können
- Ansatz: Implementierung der `invokedynamic`-Instruktion *in Hardware* auf der SHAP-Plattform
  - Code-Transformation im Linker würde entfallen oder reduziert werden
  - aber: nötig werden Typüberprüfungen und dynamisches Erzeugen von Klassen zur Laufzeit → Overhead
  - dafür: dynamische Sprachen könnten auch auf SHAP laufen

Zusammenfassung

- alle neuen Java 8-Sprachfeatures auf der SHAP-Plattform implementiert
- nur Änderungen im Linker nötig
- SHAP mit Überstützung von Java 8:
  modernste eingebettete Java-Plattform mit eigenem Bytecode-Prozessor
- viele ähnliche Projekte sind im Zeitraum 2000 – 2010 entstanden und abgebrochen worden

Vielen Dank für

Ihre Aufmerksamkeit!

»Wissen schafft Brücken.«