

Übungsblatt 12

1. Erstelle einen Rechen-Server, zu dem Klassen zur Berechnung übertragen werden können. Der Client schreibt Klassen, welche ein bestimmtes Interface implementieren, die irgendwelche Berechnungen durchführen. Diese werden dann zum Server übertragen, dort ausgeführt und das Ergebnis an den Client zurückgegeben.

Das Interface für Jobs könnte so aussehen:

```
package common;

public interface Executable extends java.io.Serializable {
    Object run( Object param );
}
```

Der Server könnte folgendes Interface zur Verfügung stellen:

```
package common;
import java.rmi.*;

public interface ExeServer extends Remote {
    // name: Name der Klasse; code: bytecode der Klasse
    // param: Parameter für run; result: Ergebnis
    Object execute( String name, byte[] code, Object param )
        throws RemoteException;
}
```

2. Schreibe ein einfaches Programm, an dem man die begrenzte Orts-Transparenz von RMI sieht.
3. Gegeben sei folgende Klasse und daneben eine Übersetzung der Methode `m` nach Bytecode (einige Details weggelassen). Das Kommando `ldc` gibt eine Referenz auf die jeweilige Konstante auf den Stack.

```
class Example2 {
    void m(Object arg) {
        Object local;
        local = "Hello";
        local.concat(" World!");
        arg.concat("Ohh!");
        ...
    }
}
```

```
void m(java.lang.Object)
0:  ldc  #2; //String "Hello"
2:  astore_2
3:  aload_2
4:  ldc  #3; //String " World!"
6:  invokevirtual String.concat(...)
10: aload_1
11: ldc  #5; //String "Ohh!"
13: invokevirtual String.concat(...)
```

- A. Gibt es mit der Verwendung der lokalen Variablen `local` im Sourcecode ein Problem?
- B. Gibt es mit der Verwendung der lokalen Variablen `local` bei der Bytecode-Verifikation ein Problem?

- C. Gibt es mit der Verwendung des Parameters `arg` im Sourcecode ein Problem?
- D. Gibt es mit der Verwendung des Parameters `arg` bei der Bytecode-Verifikation ein Problem?
3. Gegeben seien folgende Klassen und Interfaces:

```
interface Iface {
    void m();
}

class Cl1 implements Iface {
    public void m() {
        System.out.println("Cl1.m");
    }
}

class Cl2 implements Iface {
    public void m() {
        System.out.println("Cl2.m");
    }
}

public class Test1 {
    public static void main( String[] args ) {
        xxx(true);
        xxx(false);
    }

    public static void xxx( boolean param ) {
        Iface iface = null;

        if( param ) {
            iface = new Cl1();
        } else {
            iface = new Cl2();
        }

        iface.m();
    }
}
```

- A. Welcher Typ wird bei der Bytecode-Verifikation für die lokale Variable `iface` der Methode `xxx` errechnet?
- B. Wann wird überprüft, ob der Aufruf der Methode `m` auf `iface` gültig ist: beim Kompilieren, bei der Bytecode-Verifikation oder vor dem tatsächlichen Methodenaufruf?

4. Klausurbeispiel vom letzten Jahr!

Im Folgenden zeigen wir jeweils zuerst Java Sourcecode und dann eine mögliche Kodierung der selben Funktionalität direkt in Bytecode.

Begründen Sie Ihre Antwort jeweils ausführlich!

- a. Ist folgender Java Code typkorrekt?

```
int v = 5;  
v = this;
```

- b. Akzeptiert der Bytecode-Verifier den Bytecode?

```
01: iconst 5  
02: istore 1  
03: aload 0  
04: astore 1
```

Erklären Sie den Ablauf der Verifikation Schritt für Schritt.
Vergleichen Sie das Ergebnis mit der Typprüfung des Compilers.
Erläutern Sie allfällige Abweichungen.

- c. Ist folgender Java Code typkorrekt?

```
int v = 5;  
v = this;  
v = v + 1;
```

- d. Akzeptiert der Bytecode-Verifier den Bytecode?

```
01: iconst 5  
02: istore 1  
03: aload 0  
04: astore 1  
05: iload 1  
06: iconst 1  
07: iadd  
08: istore 1
```

Erklären Sie den Ablauf der Verifikation Schritt für Schritt.
Vergleichen Sie das Ergebnis mit der Typprüfung des Compilers.
Erläutern Sie allfällige Abweichungen.