Föreläsning 22

Lars-Henrik Eriksson

Föreläsningsbilderna är baserade på bilder gjorda av Tobias Wrigstad.

Under huven på JVM. Bytekod. JIT. Reflektion. Profilering.

Vad innehåller en .class-fil?

- Diverse information om klassen.
- Konstanta data som används i klassen.
- Det kompilerade metoderna och funktionerna i form av bytekod för Java Virtual Machine (JVM).
- JVM interpreterar bytekoden.
- En klassfil kompilerad på en plattform kan användas på alla andra plattformar!

Vad innehåller en .class-fil?

```
ClassFile {
  u4         magic;
  u2         minor_version;
  u2         major_version;
  u2         constant_pool_count;
  cp_info    constant_pool[constant_pool_count-1];
  u2         access_flags;
  u2         this_class;
  u2         super_class;
  u2         interfaces_count;
  u2         interfaces[interfaces_count];
  u2         fields_count;
  field_info    fields[fields_count];
  u2         methods_count;
  method_info   methods[methods_count];
  u2          attributes_count;
  attribute_info attributes[attributes_count];
}
```

JVM:en är en stackmaskin

```
public class Greeter {
    public Greeter(String s) { … }
    …
}

new Greeter(”Hello”)
```

- En stack med varvat data och operationer

  Varje operation pushar eller poppar något från stacken

  Exempel: konstruera ett nytt objekt

  ```
  new Greeter // klasser, konstanter, etc. är index in i konstantpoolen
  dup
  ldc ”Hello”
  invokespecial
  ```

- En virtuell maskin har en väldigt enkel struktur

  Datastrukturer som representerar stacken, lokala variabler och heapen.

  En loop som läser maskininstruktionen från toppen av stacken, utför den, poppar vad den behöver och pushar nya instruktioner som resultat

Inspektera en .class-file [Hello.class]

Ändrat i och med Java 9 (Ändras igen i Java 12)

```
public class Hello {
    public void greet(String s) {
        System.out.println("Hello, " + s + "!");
    }
}
```

Kompileras i Java 8 som om det hade stått.....

```
public class Hello {
    public void greet(String s) {
        System.out.println(
          new StringBuilder().append("Hello, ").append(s).append("!").toString()
        );
    }
}
```

Inspektera en .class-file [javap -c Hello]

Ändrat i och med Java 9 (Ändras igen i Java 12)

```
Compiled from "Hello.java"
public class Hello {
  Hello();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public void greet(java.lang.String);
    Code:
       0: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: new           #3                  // class java/lang/StringBuilder
       6: dup
       7: invokespecial #4                  // Method StringBuilder."<init>":()V
      10: ldc           #5                  // String Hello,
      12: invokevirtual #6                  // Method StringBuilder.append:(LString;)LStringBuilder;
      15: aload_1
      16: invokevirtual #6                  // Method StringBuilder.append:(LString;)LStringBuilder;
      19: ldc           #7                  // String !
      21: invokevirtual #6                  // Method StringBuilder.append:(LString;)LStringBuilder;
      24: invokevirtual #8                  // Method StringBuilder.toString:()LString;
      27: invokevirtual #9                  // Method PrintStream.println:(LString;)V
      30: return
}
```

Inspektera en .class-file [Hello1.class]

OBS!
Gällde till
September 2017

```
public class Hello1 {
    public void greet(String s) {
        StringBuilder sb = new StringBuilder();
        sb.append("Hello, ");
        sb.append(s);
        sb.append("!");
        System.out.println(sb.toString());
    }
}
```

Ändrat i
och med Java 9
(Ändras igen i Java 12)

```
Compiled from "Hello1.java"
public class Hello1 {
  public Hello1();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public void greet(java.lang.String);
    Code:
       0: new           #2                  // class java/lang/StringBuilder
       3: dup
       4: invokespecial #3                  // Method java/lang/StringBuilder."<init>":()V
       7: astore_2
       8: aload_2
       9: ldc           #4                  // String Hello,
      11: invokevirtual #5                  // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
      14: pop
      15: aload_2
      16: aload_1
      17: invokevirtual #5                  // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
      20: pop
      21: aload_2
      22: ldc           #6                  // String !
      24: invokevirtual #5                  // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
      27: pop
      28: getstatic     #7                  // Field java/lang/System.out:Ljava/io/PrintStream;
      31: aload_2
      32: invokevirtual #8                  // Method java/lang/StringBuilder.toString:()Ljava/lang/String;
      35: invokevirtual #9                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
      38: return
}
```

Hello.java i nyare versioner av Java

```
Compiled from "Hello.java"
public class Hello {
  public Hello();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public void greet(java.lang.String);
    Code:
       0: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: aload_1
       4: invokedynamic #3,  0              // InvokeDynamic #0:makeConcatWithConstants:(String;)String;
       9: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
      12: return
}
```

Reflektion

- Ett program som har möjlighet att inspektera sig självt
- Kraftfullt framförallt i dynamiska programspråk — Java stöd är begränsat
- Kan skapa klasser, anropa metoder, etc.
- Kan inte på ett enkelt sätt

Ändra en metod/ta bort en metod/skapa en ny klass, etc.

Bättre stöd för introspektion, dvs. att manipulera det som redan finns

```java
import java.lang.reflect.*;

public class ReflectionDemo {
    public static void main(String[] args) throws Exception {
        if (args.length != 3) {
            System.out.println("Usage: java ReflectionDemo ClassName MethodName string");
            return;
        }

        Class c = Class.forName(args[0]);
        Object o = c.newInstance();
        Method m = c.getMethod(args[1], new Class[] {String.class});
        m.invoke(o, args[2]);
    }
}
```

```java
public class Hello {
    public void greet(String s) {
        System.out.println("Hello, " + s + "!");
    }
}
```

```
$ java ReflectionDemo Hello greet world
Hello, world!
```

”TUnit” - enkelt modultestprogram

```
import java.lang.reflect.*;

public class TUnit {
    public static void main(String[] args) throws Exception {
        if (args.length < 1) {
            System.out.println("Usage: java TUnit TestClass1 ... ");
            return;
        }

        for (String className : args) {
            Class c = Class.forName(className);
            Object o = c.newInstance();

            Method setup = null;
            Method tearDown = null;
            for (Method m : c.getMethods()) {
                if (m.getName().equals("setup")) setup = m;
                if (m.getName().equals("tearDown")) tearDown = m;
                if (setup != null && tearDown != null) break;
            }

            for (Method m : c.getMethods()) {
                if (m.getName().startsWith("test") && m.getParameterCount() == 0) {
                    if (setup != null) setup.invoke(o);
                    m.invoke(o);
                    if (tearDown != null) tearDown.invoke(o);
                }
            }
        }
    }
}
```

Användning av TUnit

```
public class Hello {
    public void greet(String s) {
        System.out.println("Hello, " + s + "!");
    }
    public void test() {
     Hello h = new Hello();
     h.greet("world");
    }
}
```

```
$ java TUnit Hello
Hello, world!
```

JIT-kompilering till maskinkod

- Vi har beskrivit interpretatorloopen i stackmaskinen
- Hyfsat effektiv, men inte alls lika effektiv som maskinoptimerad kod
- Maskinoptimerad kod = plattformsspecifik
- Java är plattformsoberoende — eftersom program körs i en virtuell maskin
- JIT-kompilering är ett försök att tillfredsställa båda behov

  Kompilera bytekod till maskinkod under körning

  Kompilering tar tid — för att de skall bli en prestandavinst måste vi kompilera selektivt, bara kod som är ”het”

  Ytterligare problem: kod laddas in efterhand som programmet körs

- `-Djava.compiler=NONE` stänger av JIT-kompilering, pröva på intensivt program för att se skillnad

Rekursiv beräkning av fibonaccital

```
import java.io.*;

public class Fib {
   public static long fibonacci(long number) {
        if ((number==0) || (number==1))
           return number;
        else
           return fibonacci(number-1)+fibonacci(number-2);
    }

    public static void main(String[] args) {
        System.out.println(
                    fibonacci(Integer.parseInt(args[0])));
    }
}
```

Med JIT-kompilering påslagen (default)

```
$ time java Fib 40
102334155

real 0m0.521s
user 0m0.484s
sys  0m0.036s
```

Med JIT-kompilering avstängd

```
$ time java -Djava.compiler=NONE Fib 40
102334155

real 0m19.772s
user 0m19.492s
sys  0m0.134s
```

With JIT compilation turned on

```
$ time java Fib 40
102334155

real 0m0.521s
user 0m0.484s
sys  0m0.036s
```

With JIT compilation turned OFF

```
$ time java -Djava.compiler=NONE Fib 40
102334155

real 0m19.772s
user 0m19.492s
sys  0m0.134s
```

Hur profilerar man ett Java-program?

- JVM:en har utmärkt stöd för telemetri.
- Andra processer – t.ex. profilerare – kan läsa av vad som händer under körningen.
- Ingen instrumentering av koden behövs för profilering som det behövdes för C.
- Det går förstås i alla fall långsammare att köra ett program med profilering eftersom JVM måste skicka ut informationen om körningen.

- Det finns flera profilerare. Vi föreslår VisualVM (`https://visualvm.github.io`)
- VisualVM kör under MacOS, Windows och Linux.
- Enkel installation enligt anvisningar på webbsidan.
- Dokumentation:
  `https://visualvm.github.io/documentation.html` och
  `https://visualvm.github.io/startupprofiler.html`

Exempelprofilering med VisualVM

```
class Bar {
    static void foo3() {int i;} // If the body is empty, then
                                // Java will optimise away foo3!
    static void foo2() {
        for (int i=0; i<100; i++) {
            foo3();
        }
    }

    static void foo() {
        for (int i=0; i<100; i++) {
            foo2();
        }
    }

    public static void main(String [] s) {
        for (int i=0; i<100; i++) {
            foo();
        }
    }
}
```

Total tid i foo

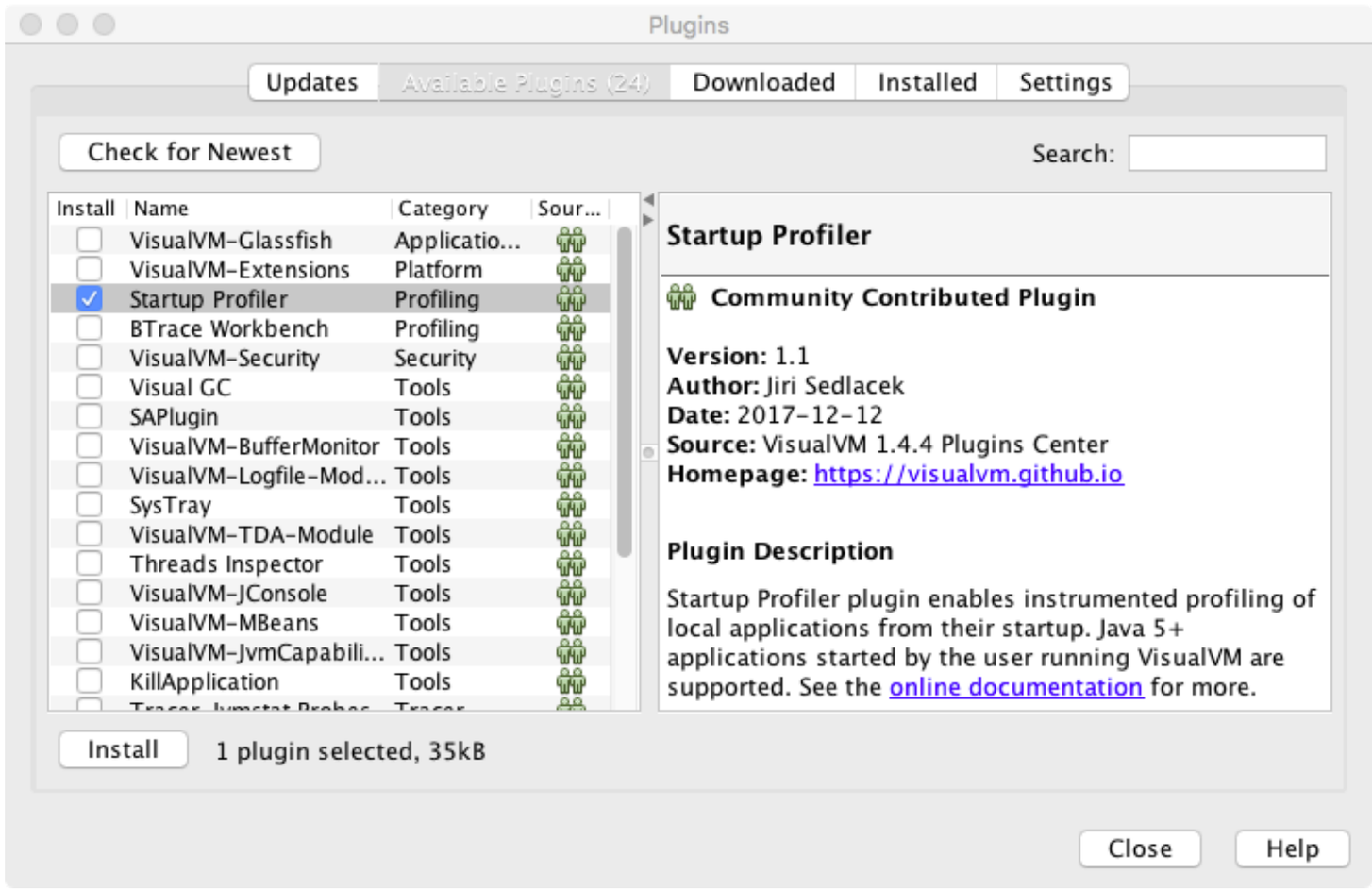
Varav tid i foo2 som anropas av foo

Varav tid i koden i definitionen av foo

Name	Total Time		Total Time (CPU)		Invocations
main	1 270 ms	(100 %)	1 239 ms	(100 %)	1
Bar.**main** (String[])	1 270 ms	(100 %)	1 239 ms	(100 %)	1
Bar.**foo** ()	1 270 ms	(100 %)	1 239 ms	(100 %)	100
Bar.**foo2** ()	1 260 ms	(99,2 %)	1 230 ms	(99,2 %)	10 000
Self time	644 ms	(50,7 %)	626 ms	(50,5 %)	10 000
Bar.**foo3** ()	616 ms	(48,5 %)	603 ms	(48,7 %)	1 000 000
Self time	9,32 ms	(0,7 %)	9,33 ms	(0,8 %)	100
Self time	0,311 ms	(0 %)	0,295 ms	(0 %)	1

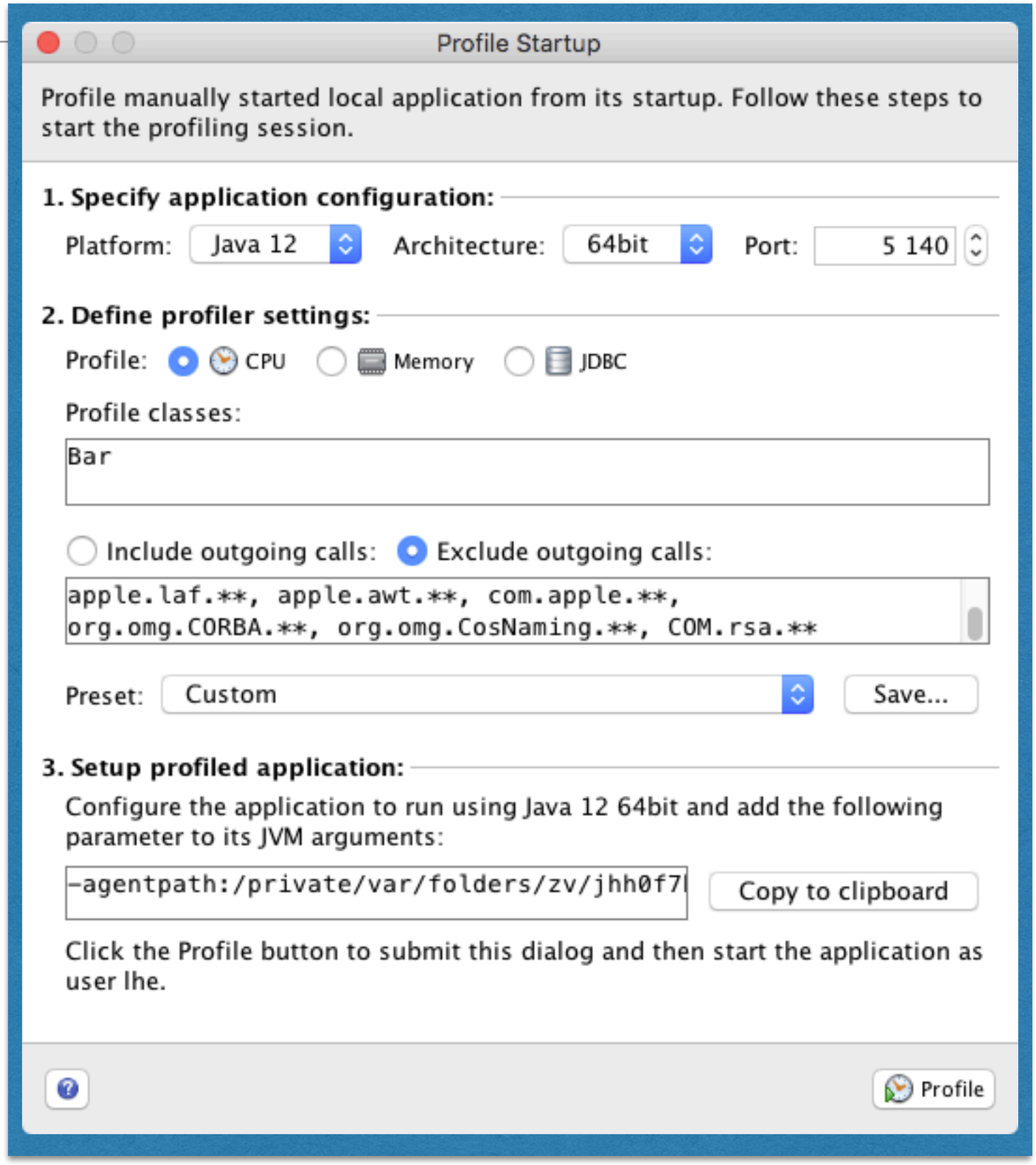
Hur man installerar Startup Profiler plugin

- Vid första körningen av VisualVM efter installationen skall man installera en plug-in.
- Starta VisualVM.
- Välj "Tools" –> "Plugins" i menyraden
- Klicka på "Available Plugins"
- Leta upp "Startup Profiler", markera kryssrutan och klicka på "Install".
- Detta behöver bara göras *en gång*.



Hur man profilerar med VisualVM

- Starta VisualVM.
- Välj "Applications" -> "Profile Startup" i menyraden
- Om det behövs, klicka på "Continue" två gånger.
- I rutan "Profile classes", skriv namnet på klasser som skall profileras -- eller "*" för alla klasser.
- Man behöver ge en *lång* parametersträng till JVM när ens program skall köra. Klicka på "Copy to clipboard" så kopieras den.
- Klicka på "Profile".
- (Fortsättning på nästa bild.)

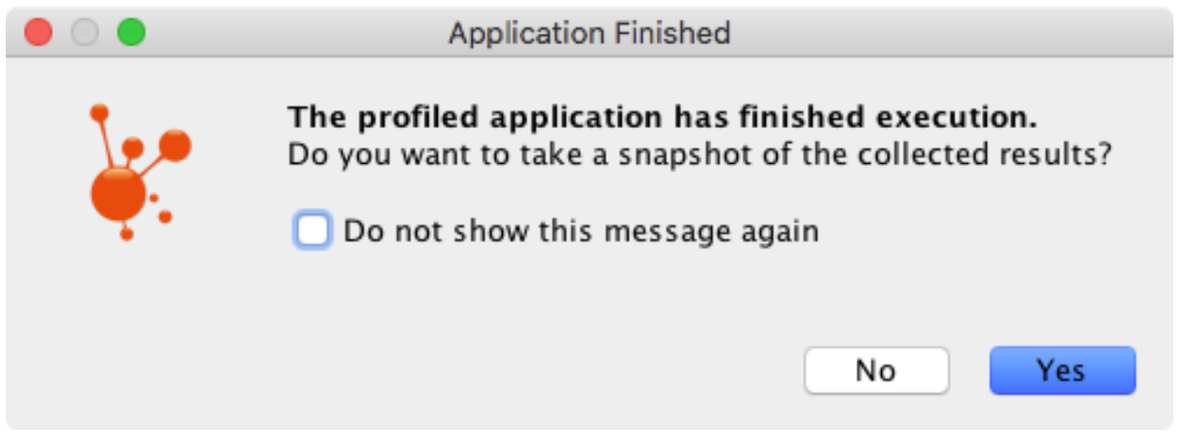


Hur man profilerar med VisualVM (forts.)

- Kör ditt program från kommandotolken, men klistra in parametern från VisualVM före klassnamnet, t.ex.

```
java -agentpath:/private/var/folders/zv/jhh0f7hn2lz2lqt5kfl83rqh0000gp/T/
AppTranslocation/4E7B68F2-4898-4BE7-96A6-797D73A3A317/d/VisualVM.app/
Contents/Resources/visualvm/profiler/lib/deployed/jdk16/mac/
libprofilerinterface.jnilib=/private/var/folders/zv/
jhh0f7hn2lz2lqt5kfl83rqh0000gp/T/AppTranslocation/
4E7B68F2-4898-4BE7-96A6-797D73A3A317/d/VisualVM.app/Contents/Resources/
visualvm/profiler/lib,5140 Bar
```

- Medan programmet kör så visar VisualVM kontinuerligt uppdaterad profileringsinformation (om man hinner se det).
- När programmet kört färdigt så frågar VisualVM

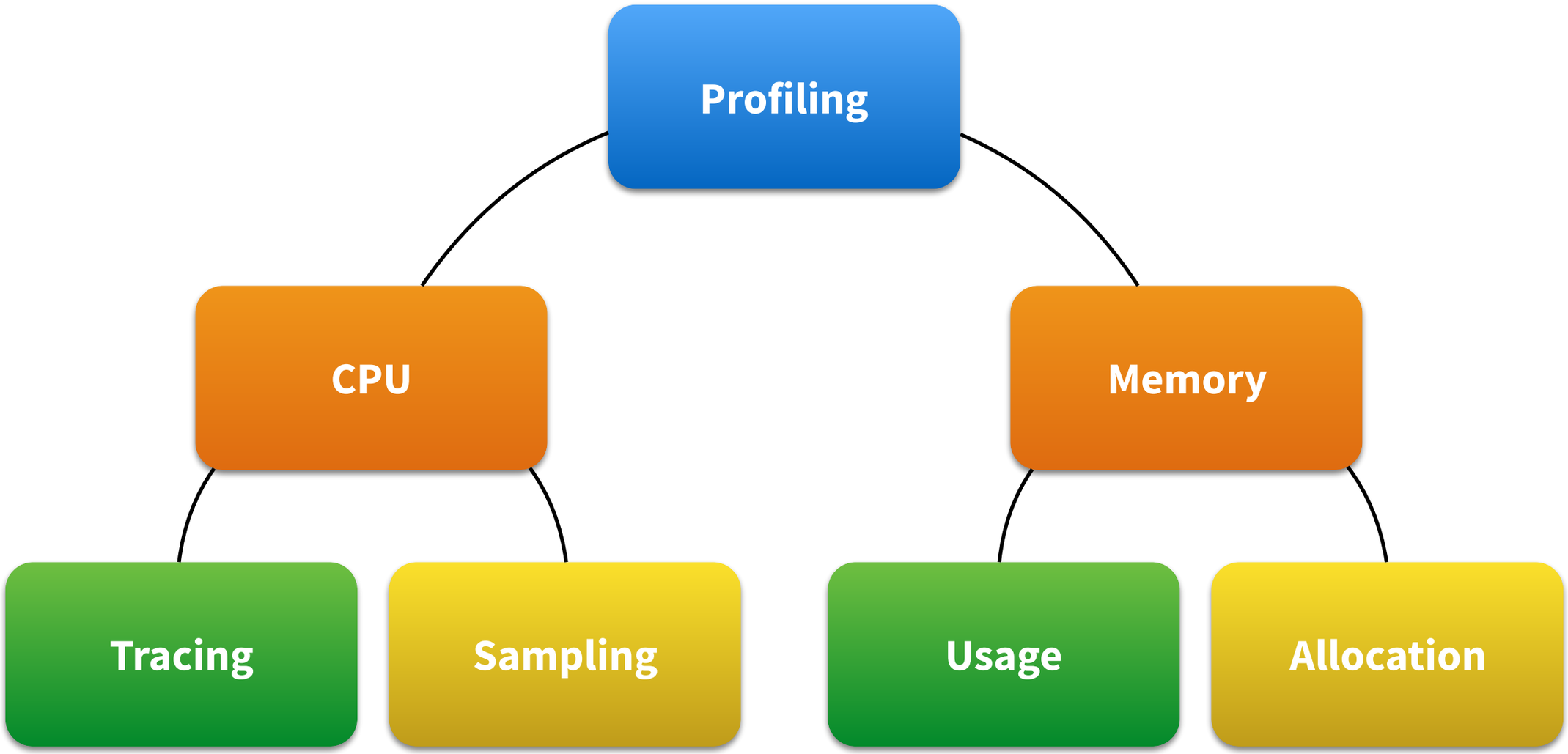


- *Tryck "Yes"!*
- Profileringen är klar!

Andra sätt att använda VisualVM

- VisualVM kan profilera Java-program som redan kör. Det kan vara praktiskt med interaktiva program som man startar först och sedan kontinuerligt profilerar.
- Under "Local" i rutan längst till vänster visas alla pågående Java-processer på maskinen.
- Dubbelklicka en process för att få information om den.
- Se vidare dokumentationen!

Typer av profilering



Sampling vs. Profiling

- Sampling: läs med jämna mellanrum (t.ex. var 10 ms) av vad programmet gör precis då och samla statistik

  Inte komplett, men mycket tidseffektivt

- Profiling: stoppa in telemetri i koden som rapporterar varje användande

  Komplett, mycket overhead, påverkar körtiden negativt

- Använd båda!

  Sampling är ofta en bättre teknik om CPU-prestanda är ditt mål

Slutord bytekod, JIT, profilering

- I vilken utsträckning måste man förstå JIT-kompilering och bytekod?

  Det beror på vad man gör — men framförallt måste man förstå vad som sker under huven om man någon gång råkar ut för ett program som inte presterar bra

- Försök inte hjälpa JVM:en att generera bra bytekod

  Den är optimerad för ”vanliga dödligas” kod

- Optimera aldrig utan att profilera fram vad som skall optimeras!

  Vad är representativ indata/omständigheter för programmet?

- Verifiera att en optimering inte är en falsk optimering genom att profilera igen!

- Undvik att optimera tills det inte är hållbart längre att inte göra det!