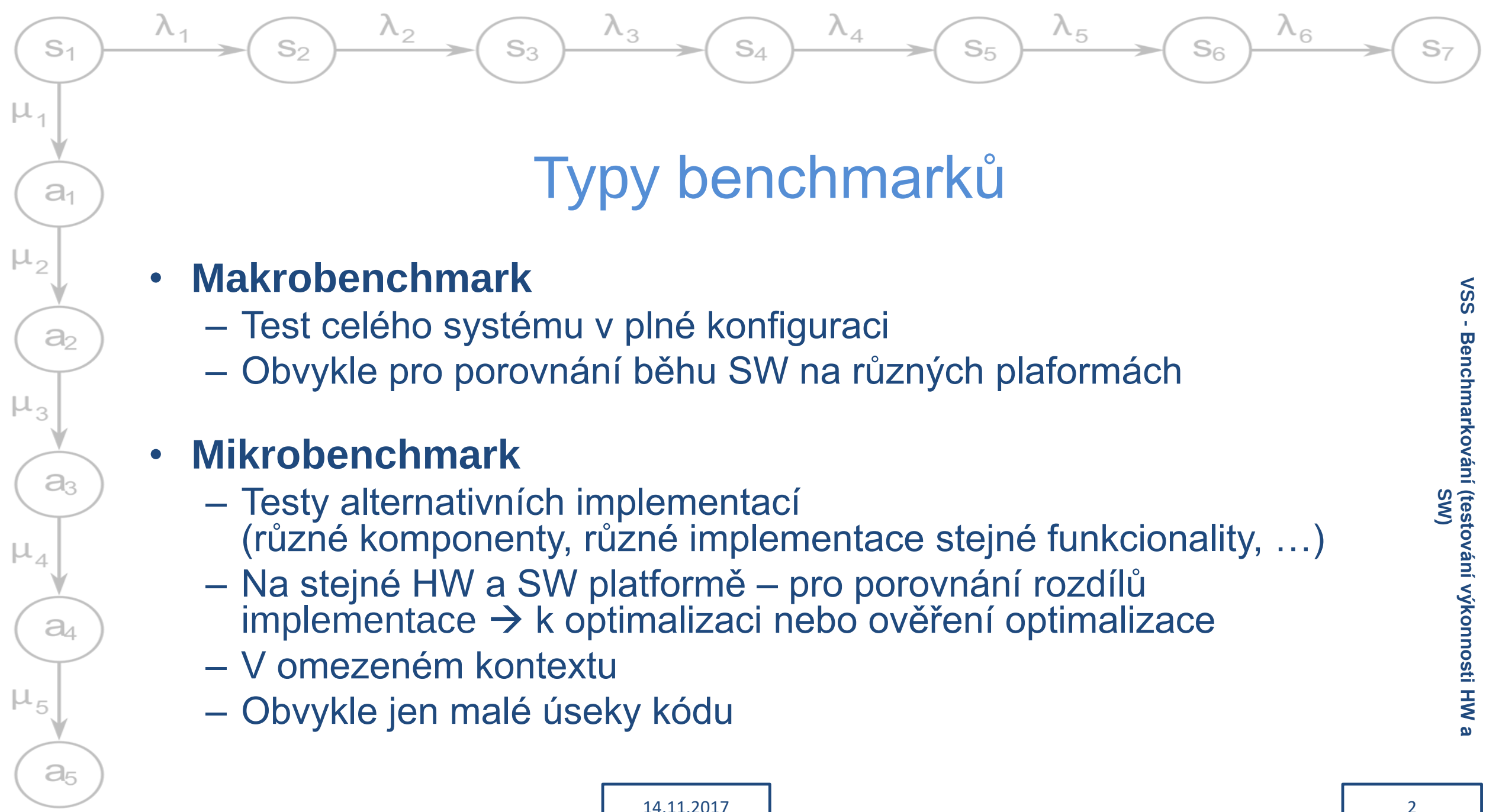
Java Microbenchmark Harness

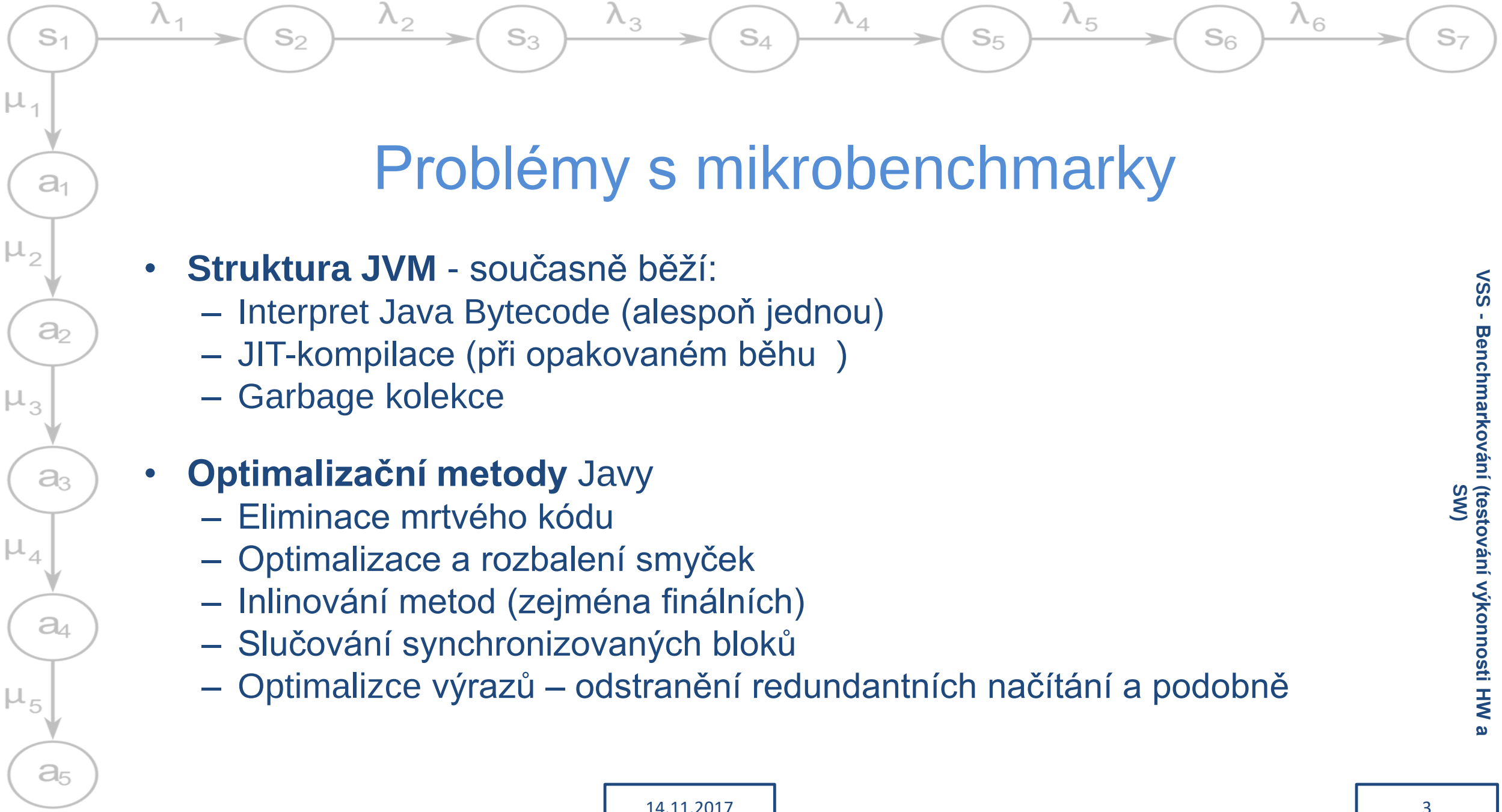
Knihovna pro tvorbu microbenchmarků

Richard Lipka



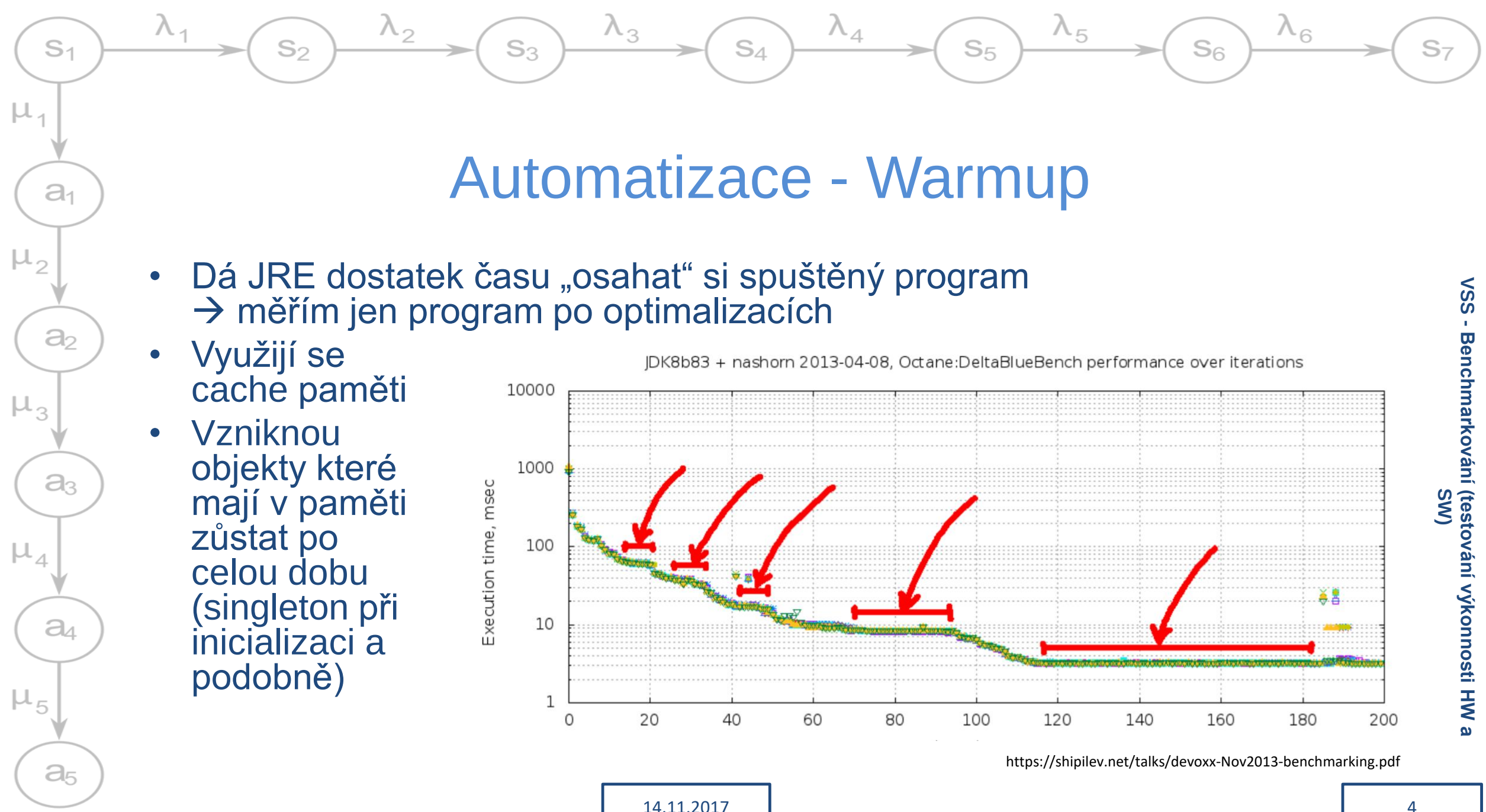
Typy benchmarků

- **Makrobenchmark**
  - Test celého systému v plné konfiguraci
  - Obvykle pro porovnání běhu SW na různých plaformách
- **Mikrobenchmark**
  - Testy alternativních implementací (různé komponenty, různé implementace stejné funkcionality, …)
  - Na stejné HW a SW platformě – pro porovnání rozdílů implementace → k optimalizaci nebo ověření optimalizace
  - V omezeném kontextu
  - Obvykle jen malé úseky kódu



Problémy s mikrobenchmarky

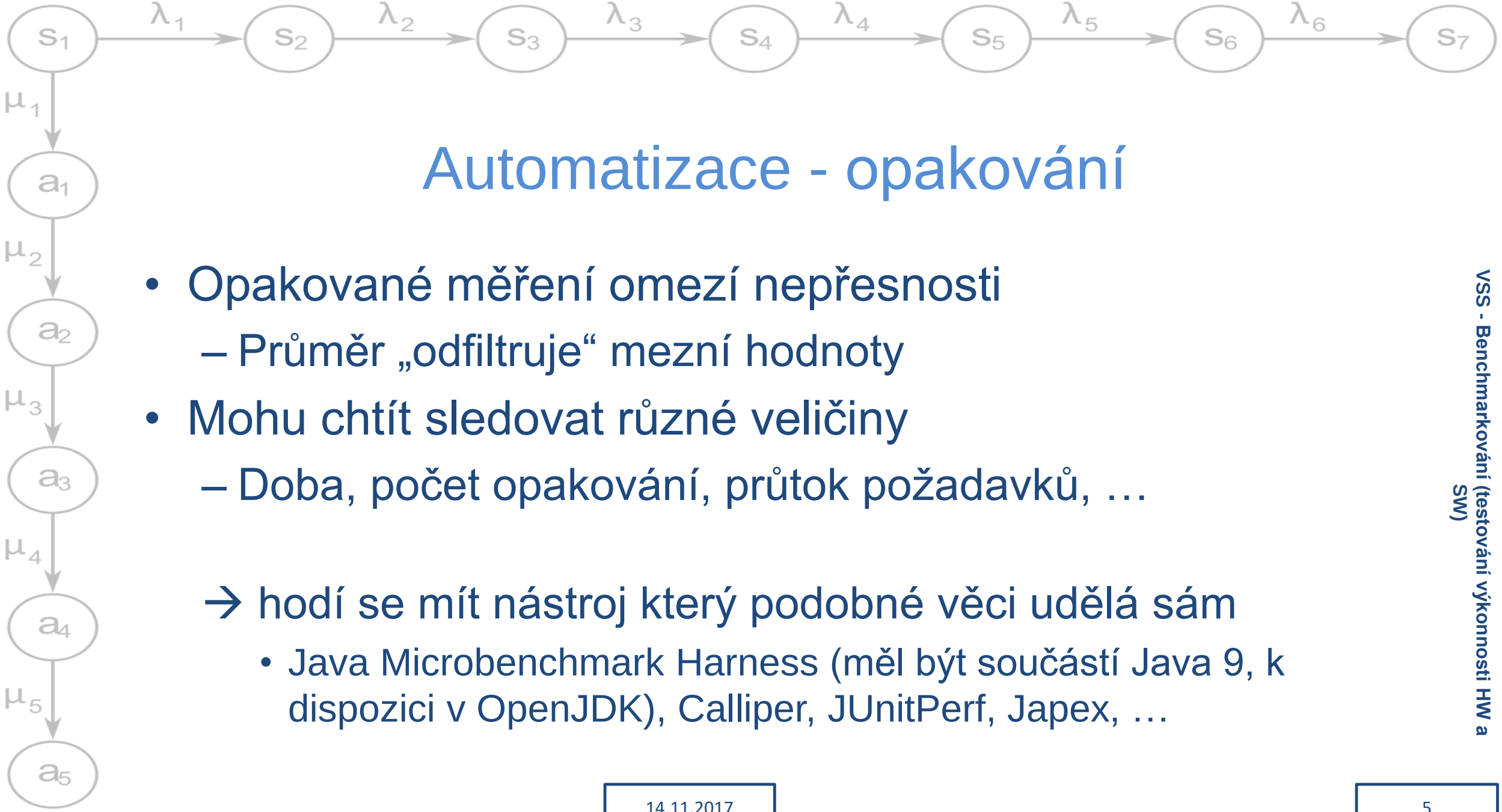
- **Struktura JVM** - současně běží:
  - Interpret Java Bytecode (alespoň jednou)
  - JIT-kompilace (při opakovaném běhu)
  - Garbage kolekce
- **Optimalizační metody** Javy
  - Eliminace mrtvého kódu
  - Optimalizace a rozbalení smyček
  - Inlinování metod (zejména finálních)
  - Slučování synchronizovaných bloků
  - Optimalizce výrazů – odstranění redundantních načítání a podobně



Automatizace - Warmup

- Dá JRE dostatek času „osahat“ si spuštěný program → měřím jen program po optimalizacích
- Využijí se cache paměti
- Vzniknou objekty které mají v paměti zůstat po celou dobu (singleton při inicializaci a podobně)

https://shipilev.net/talks/devoxx-Nov2013-benchmarking.pdf

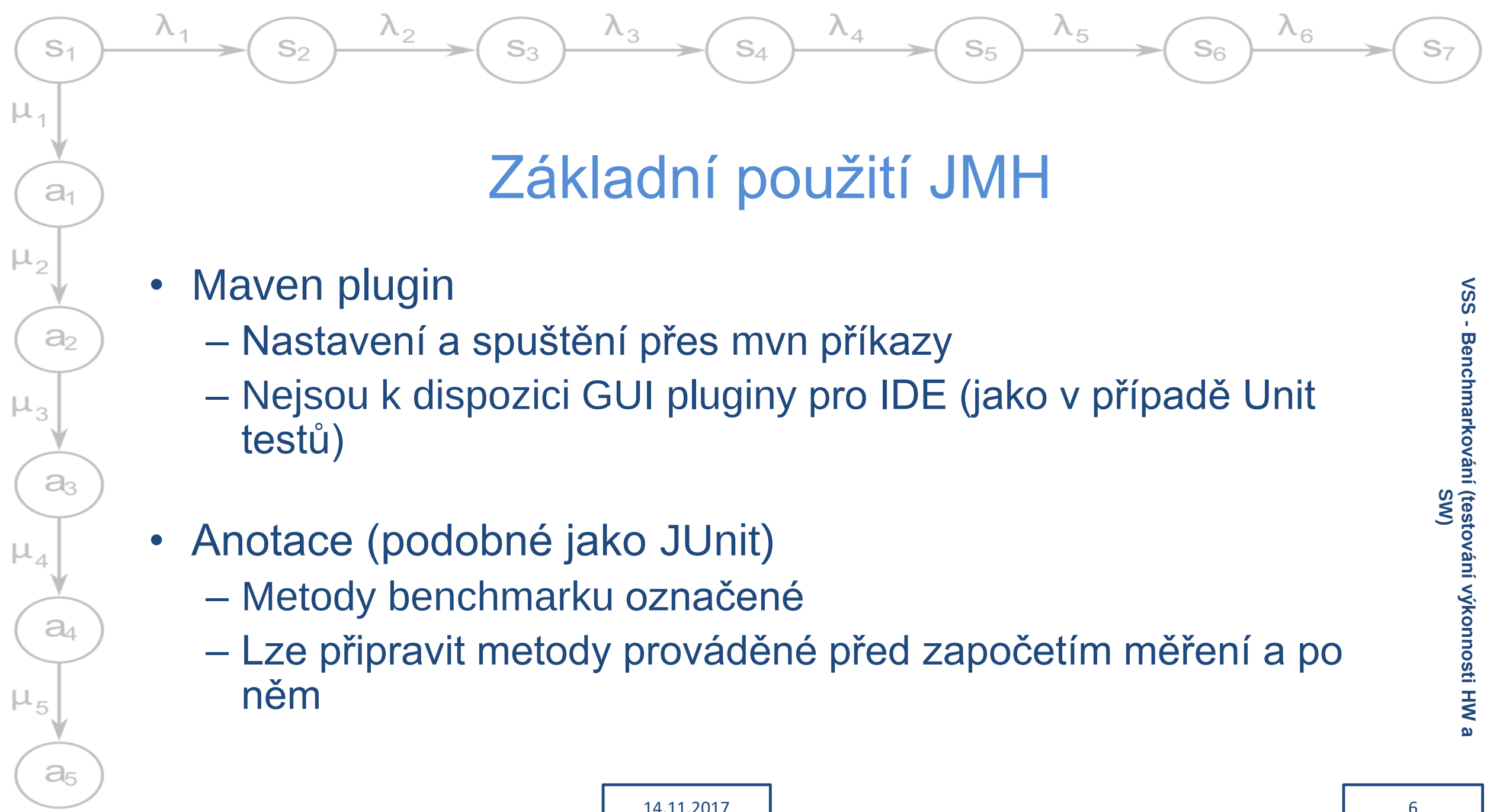


Automatizace - opakování

- Opakované měření omezí nepřesnosti
  - Průměr „odfiltruje“ mezní hodnoty
- Mohu chtít sledovat různé veličiny
  - Doba, počet opakování, průtok požadavků, …

→ hodí se mít nástroj který podobné věci udělá sám

- Java Microbenchmark Harness (měl být součástí Java 9, k dispozici v OpenJDK), Calliper, JUnitPerf, Japex, …



Základní použití JMH

- Maven plugin
  - Nastavení a spuštění přes mvn příkazy
  - Nejsou k dispozici GUI pluginy pro IDE (jako v případě Unit testů)
- Anotace (podobné jako JUnit)
  - Metody benchmarku označené
  - Lze připravit metody prováděné před započetím měření a po něm

Příprava projektu

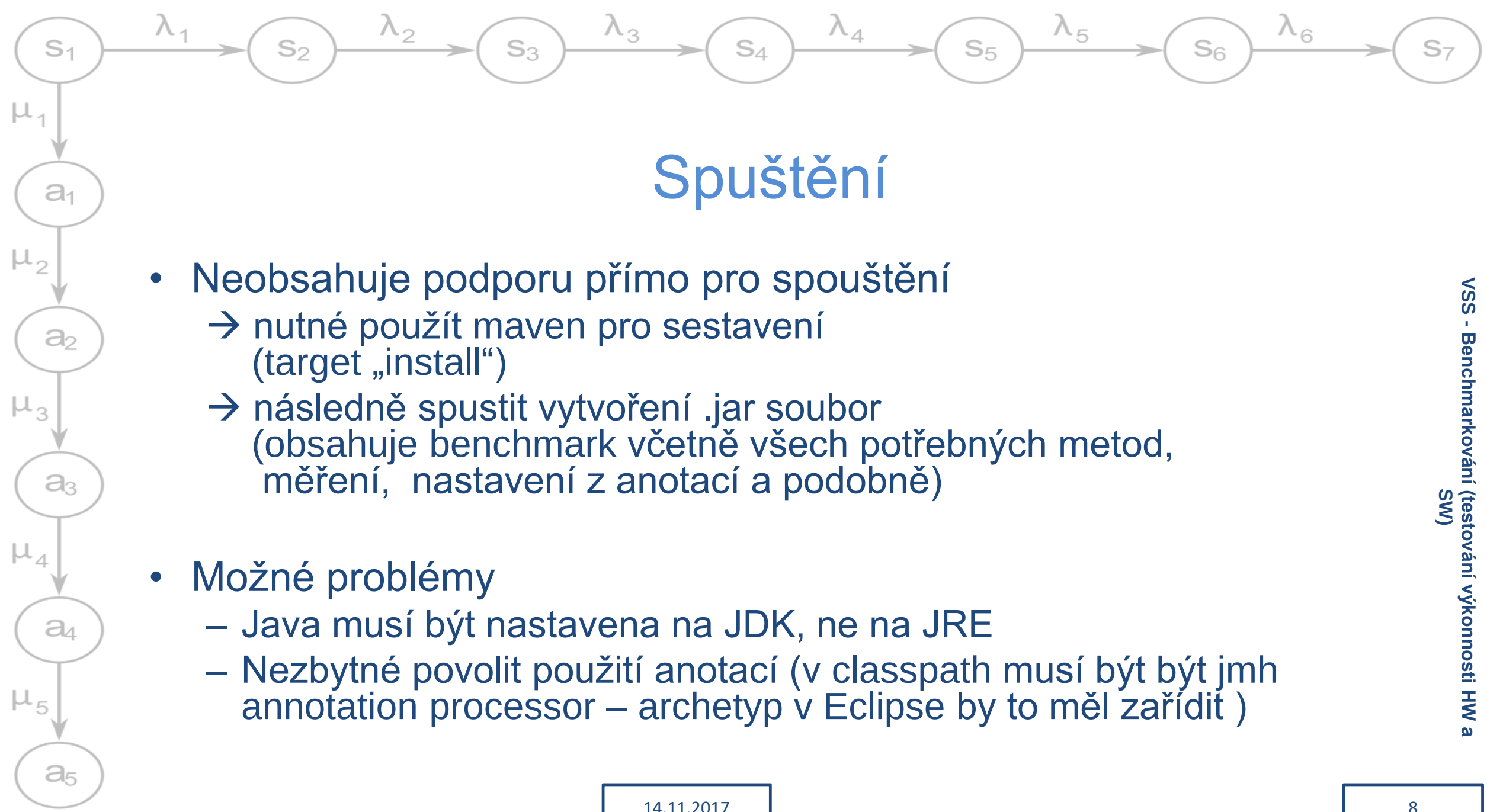
Příkaz v konzoli

```
mvn archetype:generate \
 -DinteractiveMode=false \
 -DarchetypeGroupId=org.openjdk.jmh \
 -DarchetypeArtifactId=jmh-java-
benchmark-archetype \
 -DgroupId=org.sample \
 -DartifactId=test \
 -Dversion=1.0
```

V Eclipse

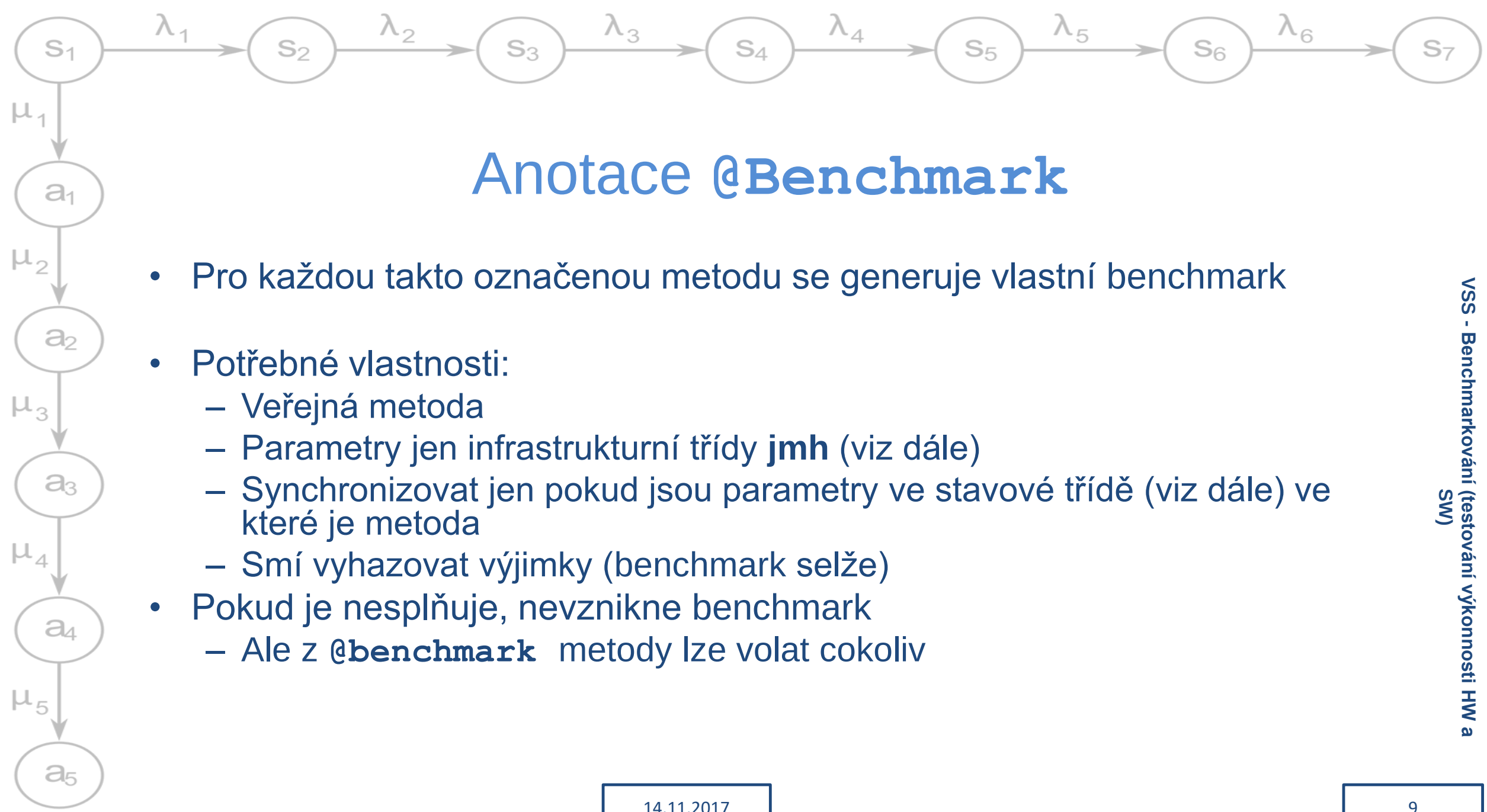
- Založit Maven projekt
- Přidat „ Central archetype catalog“
- Nastavit archetyp **`org.openjdk.jmh:jmh-${lang}-benchmark-archetype`**

- Založí v projektu třídu s prvním benchmarkem a stáhne potřebné závislosti



Spuštění

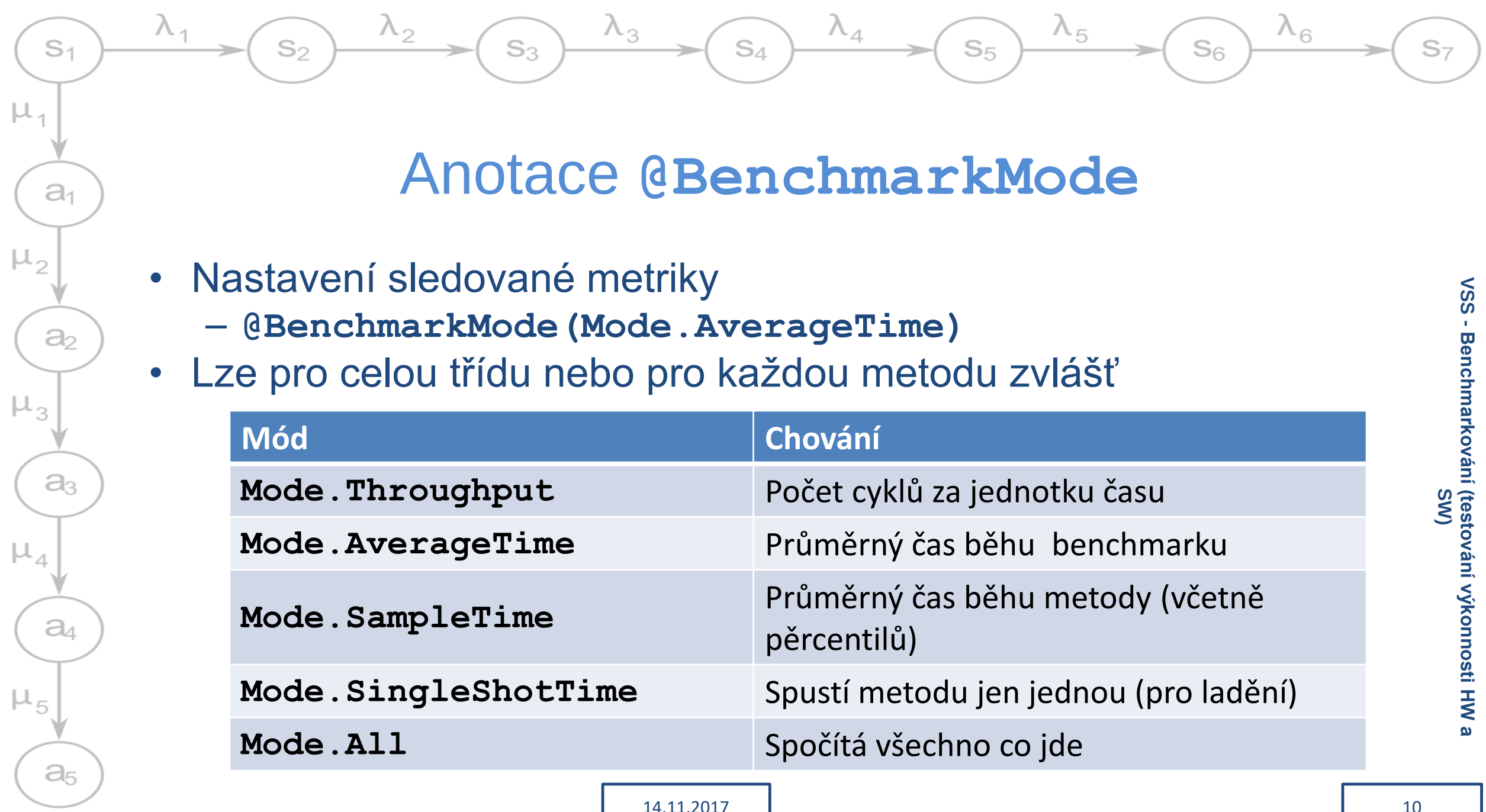
- Neobsahuje podporu přímo pro spouštění
  - → nutné použít maven pro sestavení (target „install“)
  - → následně spustit vytvoření .jar soubor (obsahuje benchmark včetně všech potřebných metod, měření, nastavení z anotací a podobně)
- Možné problémy
  - Java musí být nastavena na JDK, ne na JRE
  - Nezbytné povolit použití anotací (v classpath musí být být jmh annotation processor – archetyp v Eclipse by to měl zařídit)



Anotace `@Benchmark`

- Pro každou takto označenou metodu se generuje vlastní benchmark

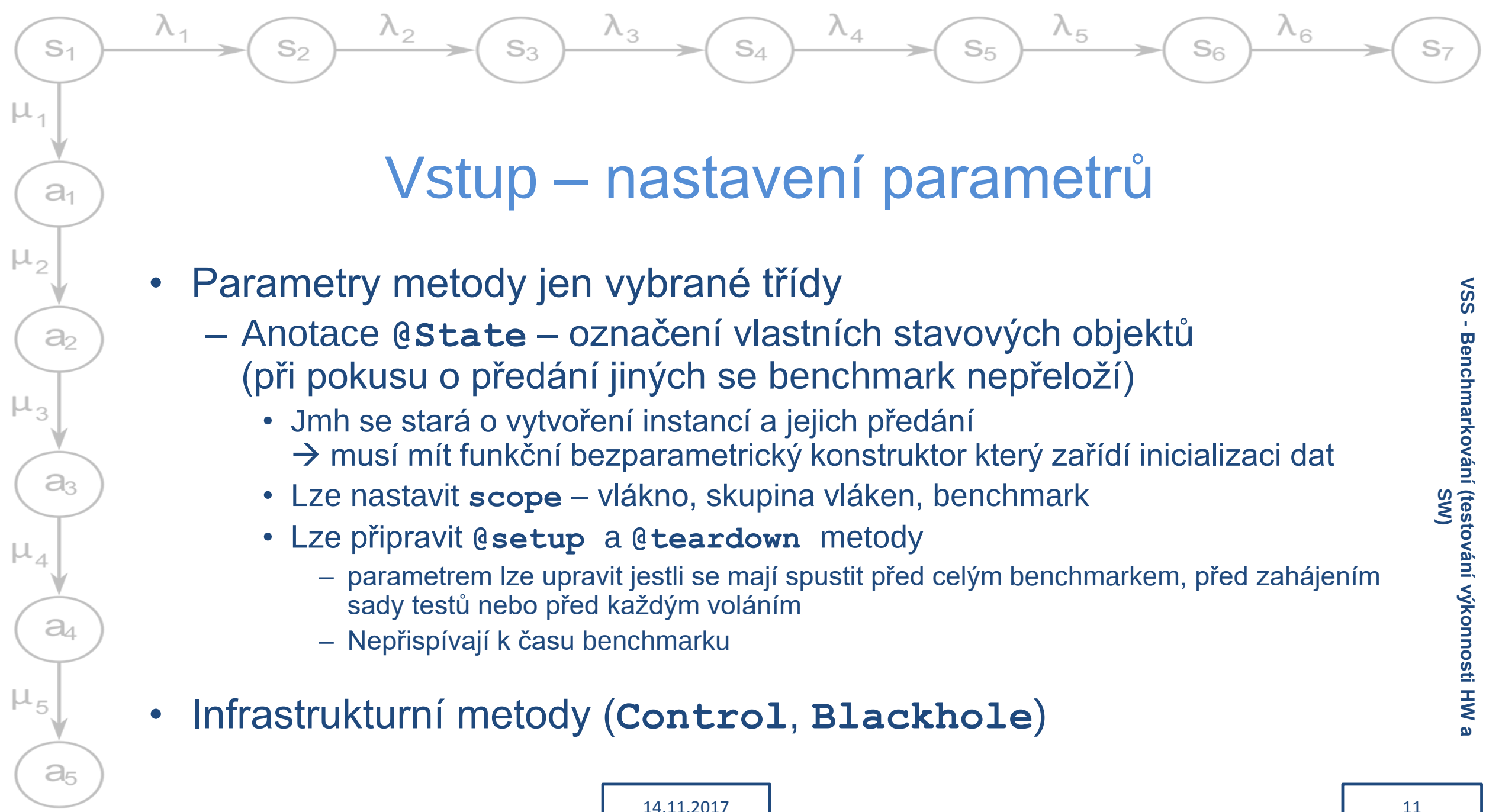
- Potřebné vlastnosti:
  - Veřejná metoda
  - Parametry jen infrastrukturní třídy **jmh** (viz dále)
  - Synchronizovat jen pokud jsou parametry ve stavové třídě (viz dále) ve které je metoda
  - Smí vyhazovat výjimky (benchmark selže)
- Pokud je nesplňuje, nevznikne benchmark
  - Ale z `@benchmark` metody lze volat cokoliv



Anotace `@BenchmarkMode`

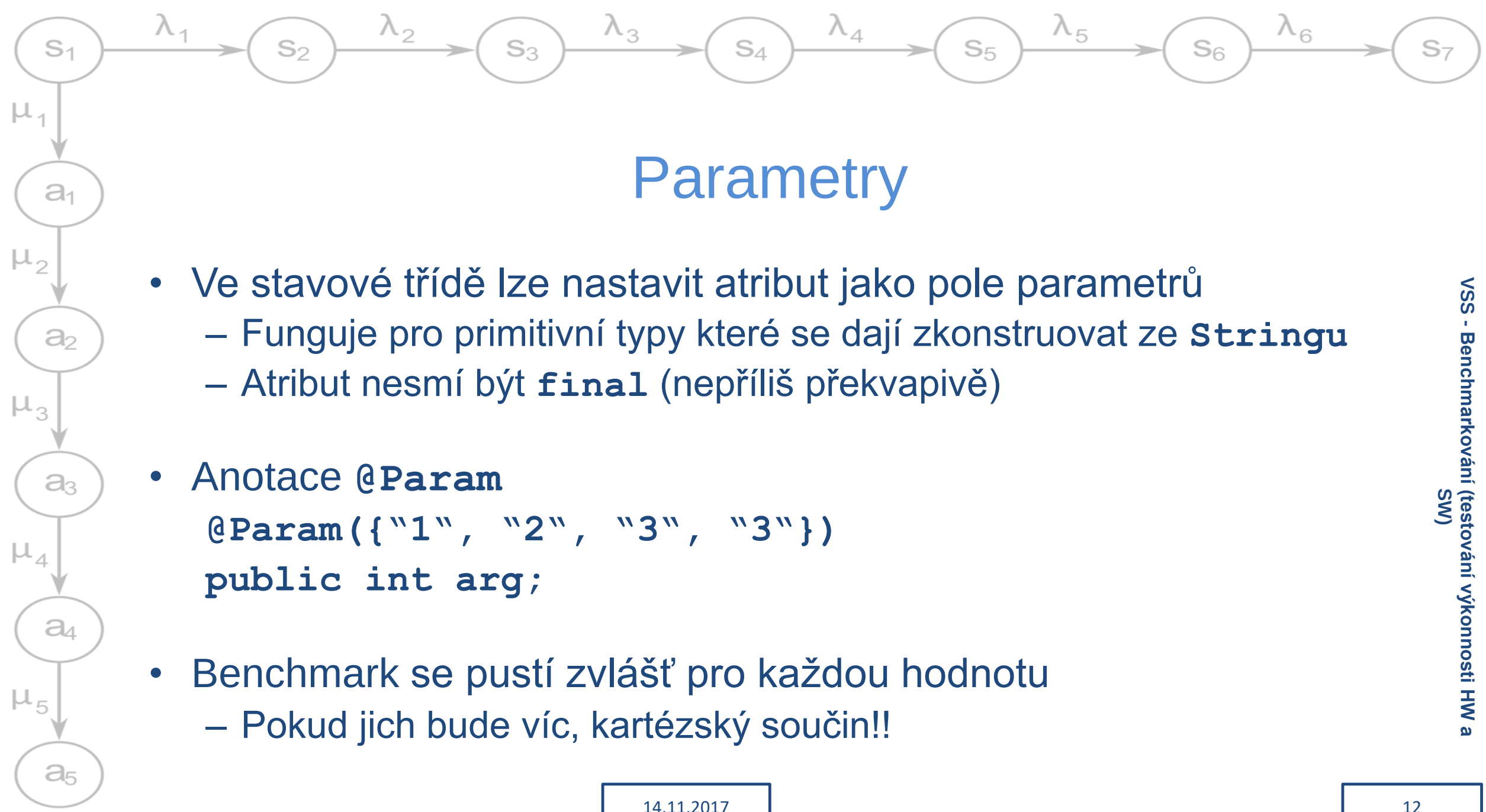
- Nastavení sledované metriky
  - `@BenchmarkMode(Mode.AverageTime)`
- Lze pro celou třídu nebo pro každou metodu zvlášť

Mód	Chování
`Mode.Throughput`	Počet cyklů za jednotku času
`Mode.AverageTime`	Průměrný čas běhu benchmarku
`Mode.SampleTime`	Průměrný čas běhu metody (včetně pěrcentilů)
`Mode.SingleShotTime`	Spustí metodu jen jednou (pro ladění)
`Mode.All`	Spočítá všechno co jde



Vstup – nastavení parametrů

- Parametry metody jen vybrané třídy
  - Anotace **`@State`** – označení vlastních stavových objektů (při pokusu o předání jiných se benchmark nepřeloží)
    - Jmh se stará o vytvoření instancí a jejich předání → musí mít funkční bezparametrický konstruktor který zařídí inicializaci dat
    - Lze nastavit **`scope`** – vlákno, skupina vláken, benchmark
    - Lze připravit **`@setup`** a **`@teardown`** metody
      - parametrem lze upravit jestli se mají spustit před celým benchmarkem, před zahájením sady testů nebo před každým voláním
      - Nepřispívají k času benchmarku
- Infrastrukturní metody (**`Control`**, **`Blackhole`**)

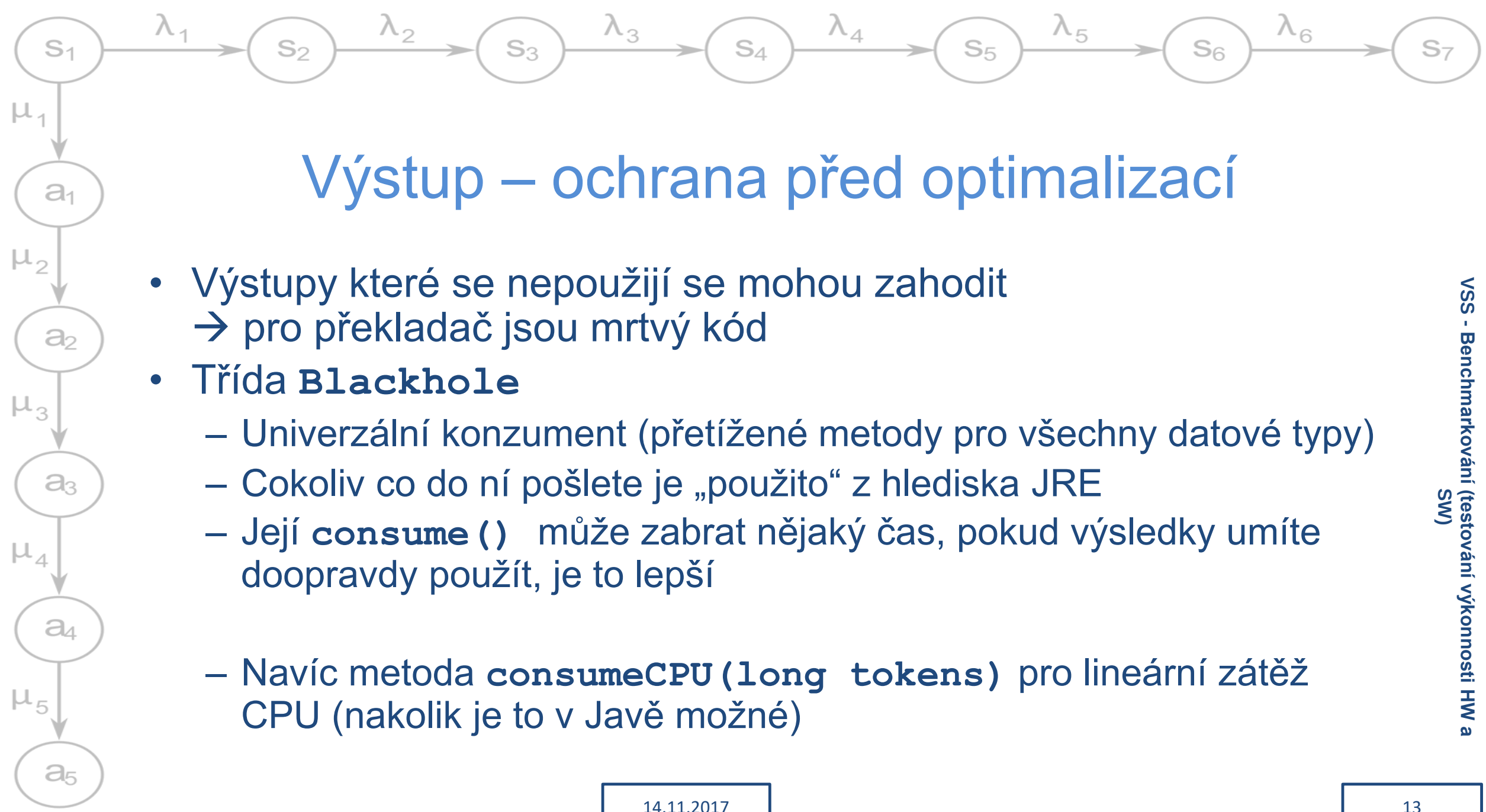


Parametry

- Ve stavové třídě lze nastavit atribut jako pole parametrů
  - Funguje pro primitivní typy které se dají zkonstruovat ze **`Stringu`**
  - Atribut nesmí být **`final`** (nepříliš překvapivě)
- Anotace **`@Param`**

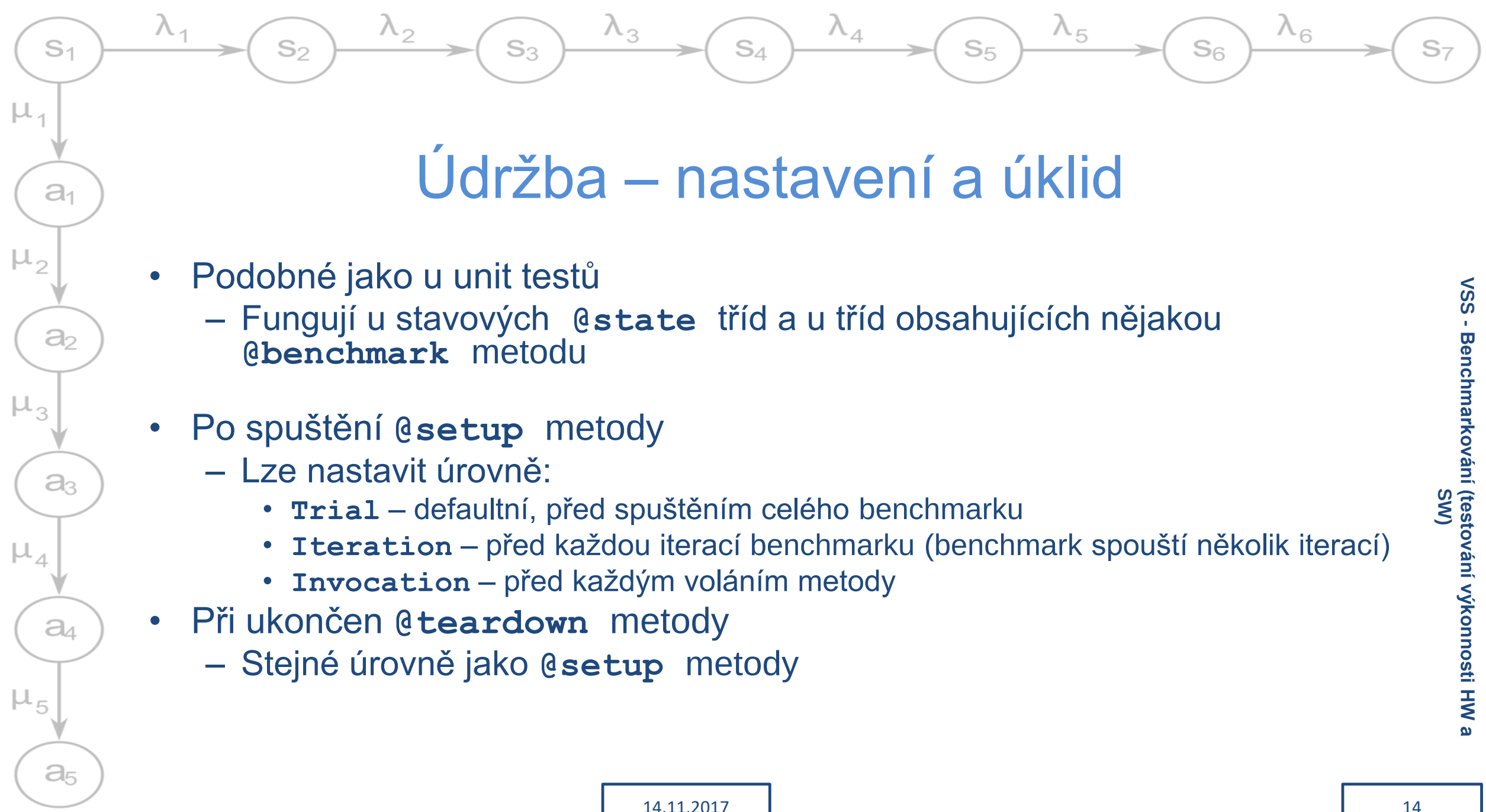
```
@Param({"1", "2", "3", "3"})
public int arg;
```

- Benchmark se pustí zvlášť pro každou hodnotu
  - Pokud jich bude víc, kartézský součin!!



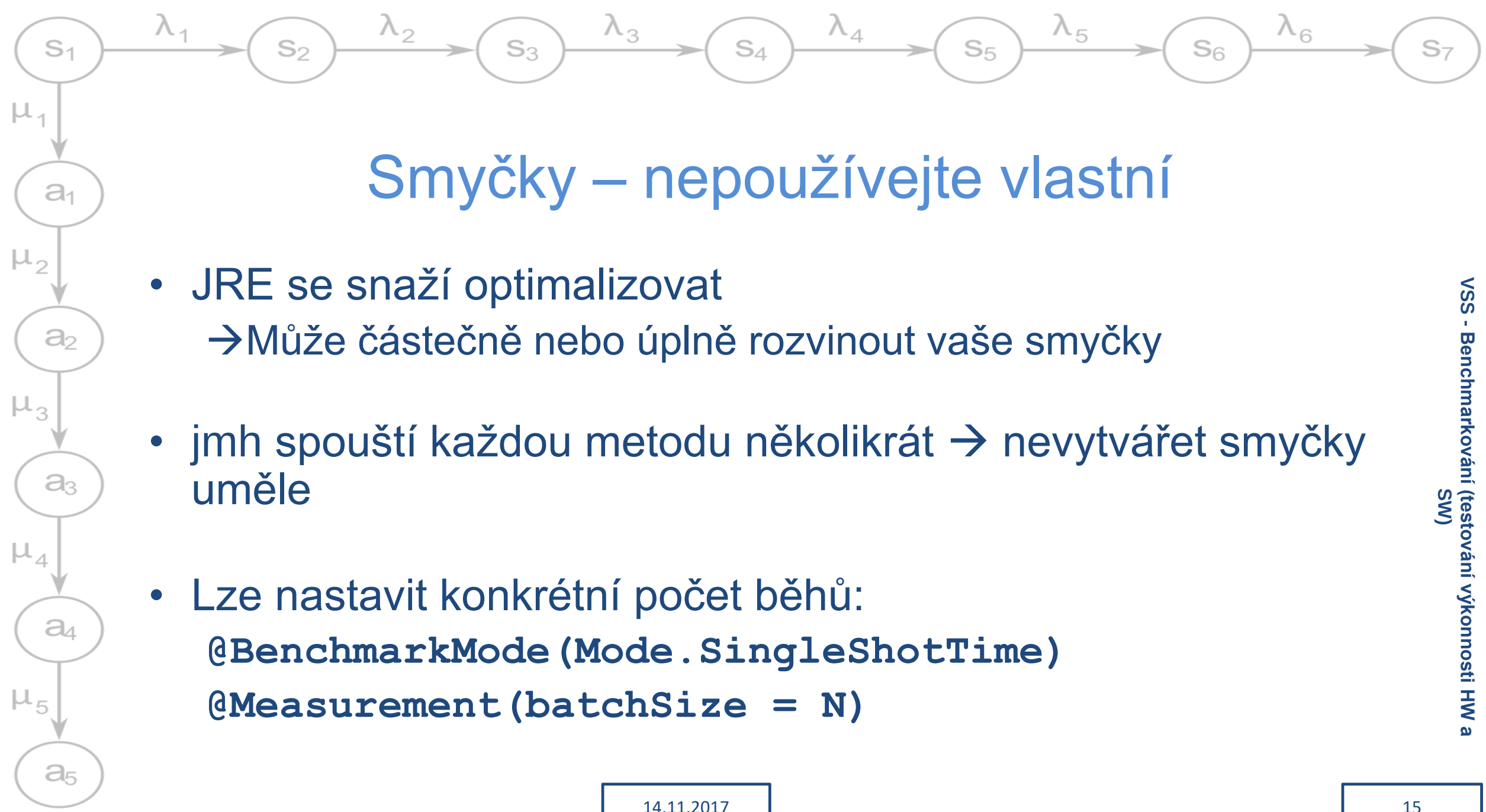
Výstup – ochrana před optimalizací

- Výstupy které se nepoužijí se mohou zahodit
  → pro překladač jsou mrtvý kód
- Třída **`Blackhole`**
  - Univerzální konzument (přetížené metody pro všechny datové typy)
  - Cokoliv co do ní pošlete je „použito“ z hlediska JRE
  - Její **`consume()`** může zabrat nějaký čas, pokud výsledky umíte doopravdy použít, je to lepší
  - Navíc metoda **`consumeCPU(long tokens)`** pro lineární zátěž CPU (nakolik je to v Javě možné)



Údržba – nastavení a úklid

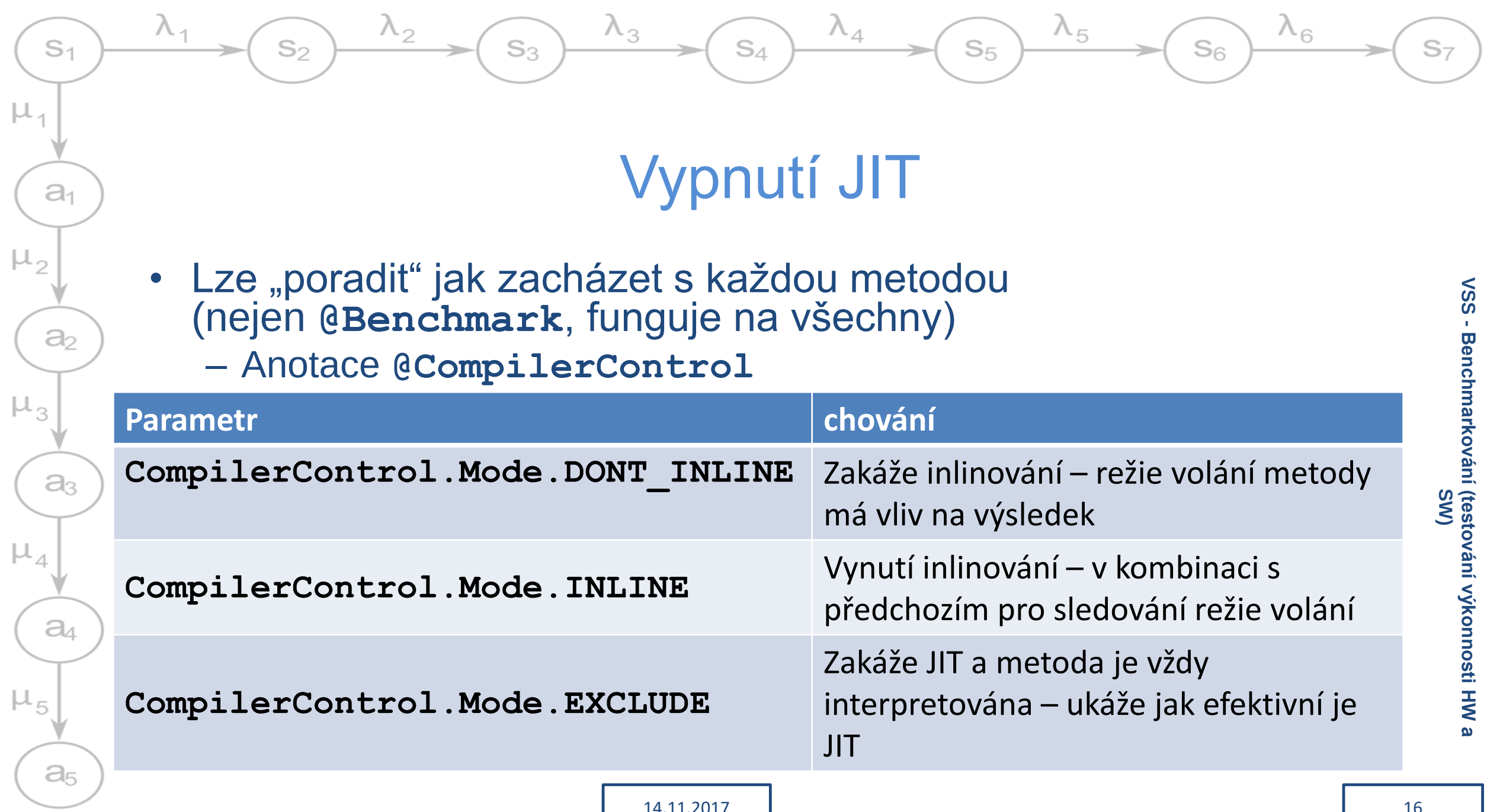
- Podobné jako u unit testů
  - Fungují u stavových `@state` tříd a u tříd obsahujících nějakou `@benchmark` metodu
- Po spuštění `@setup` metody
  - Lze nastavit úrovně:
    - `Trial` – defaultní, před spuštěním celého benchmarku
    - `Iteration` – před každou iterací benchmarku (benchmark spouští několik iterací)
    - `Invocation` – před každým voláním metody
- Při ukončen `@teardown` metody
  - Stejné úrovně jako `@setup` metody



Smyčky – nepoužívejte vlastní

- JRE se snaží optimalizovat
  →Může částečně nebo úplně rozvinout vaše smyčky
- jmh spouští každou metodu několikrát → nevytvářet smyčky uměle
- Lze nastavit konkrétní počet běhů:

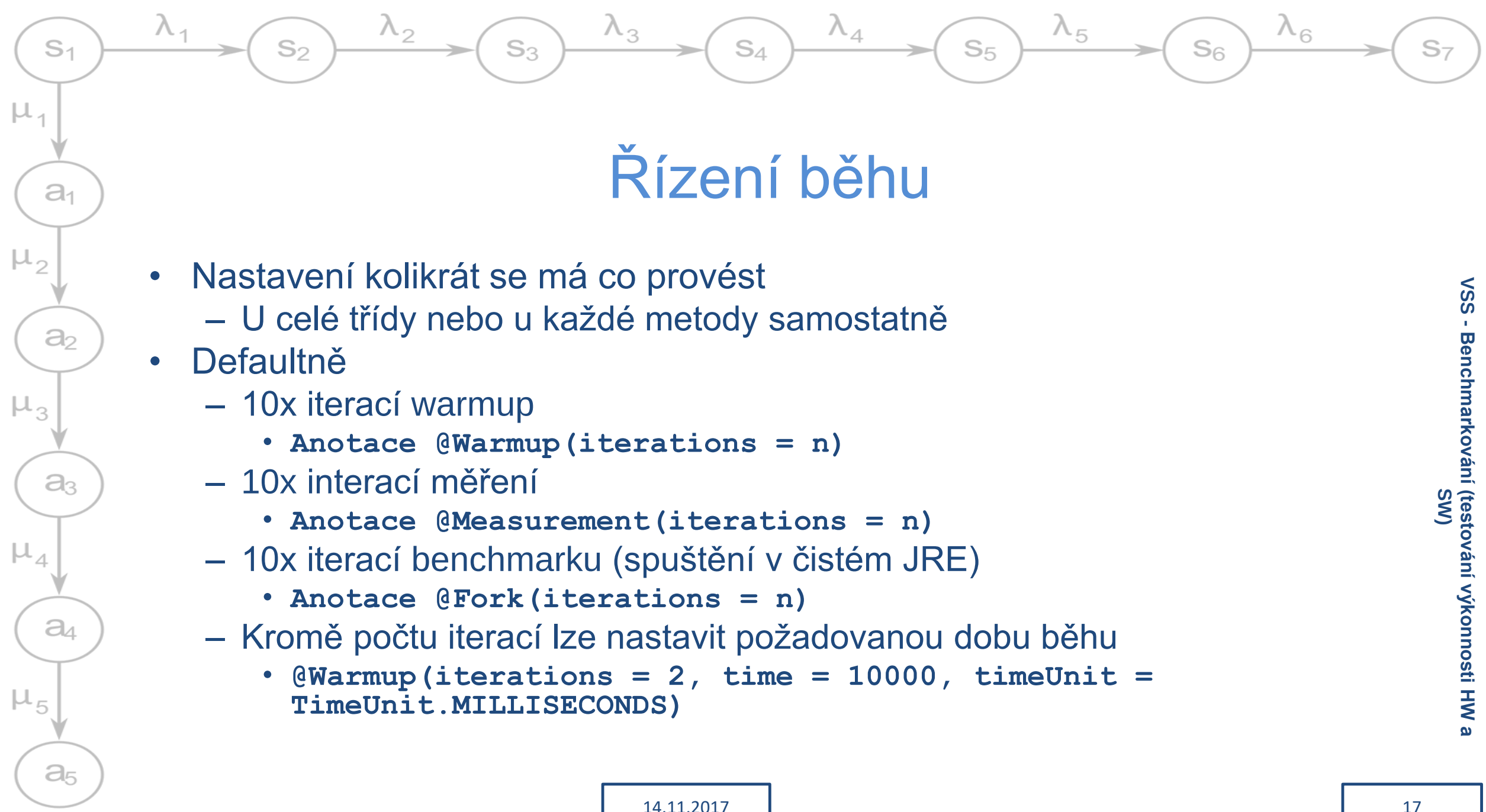
```
@BenchmarkMode(Mode.SingleShotTime)
@Measurement(batchSize = N)
```



Vypnutí JIT

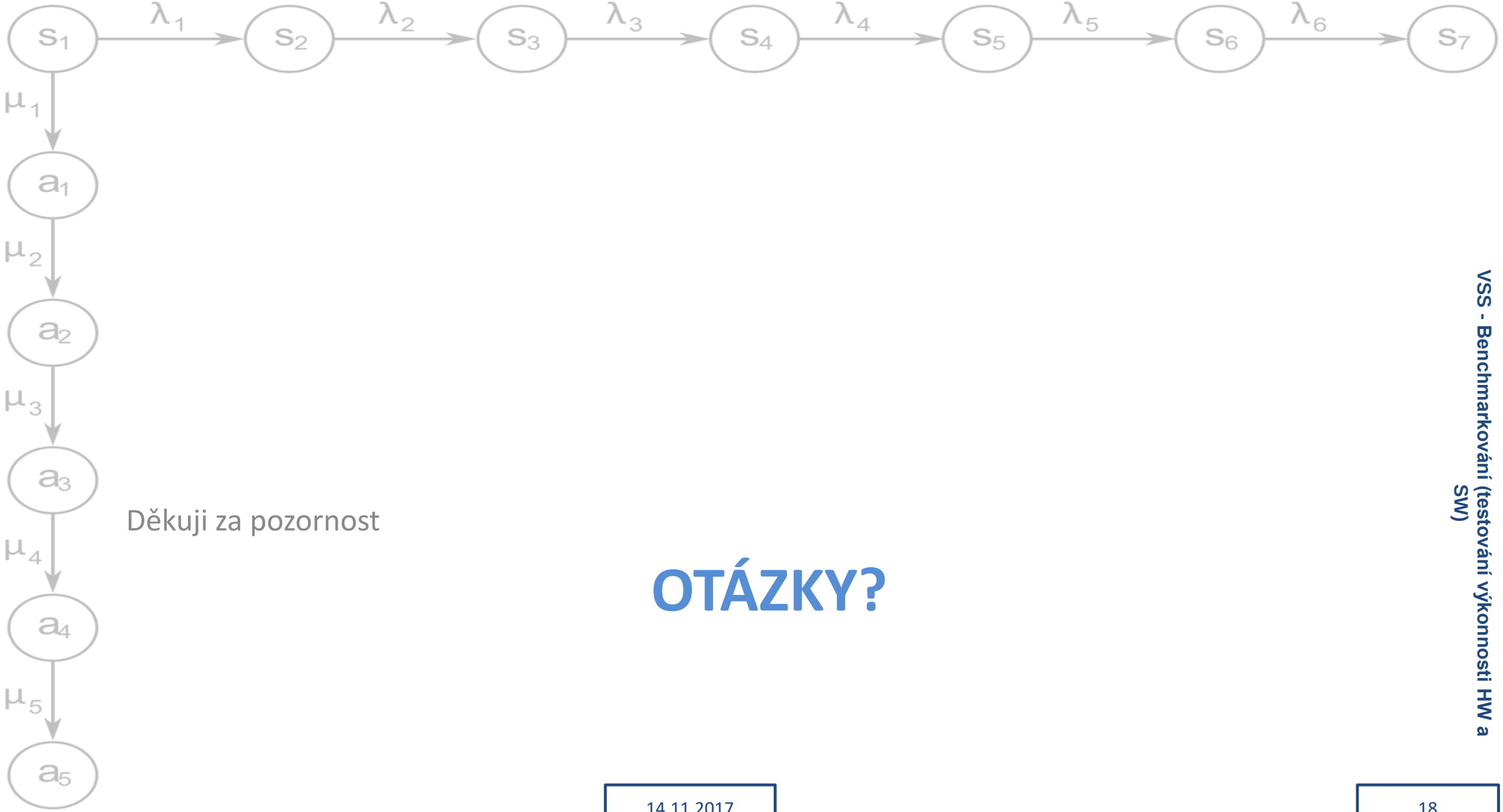
- Lze „poradit“ jak zacházet s každou metodou (nejen **`@Benchmark`**, funguje na všechny)
  - Anotace **`@CompilerControl`**

Parametr	chování
`CompilerControl.Mode.DONT_INLINE`	Zakáže inlinování – režie volání metody má vliv na výsledek
`CompilerControl.Mode.INLINE`	Vynutí inlinování – v kombinaci s předchozím pro sledování režie volání
`CompilerControl.Mode.EXCLUDE`	Zakáže JIT a metoda je vždy interpretována – ukáže jak efektivní je JIT



Řízení běhu

- Nastavení kolikrát se má co provést
  - U celé třídy nebo u každé metody samostatně
- Defaultně
  - 10x iterací warmup
    - **`Anotace @Warmup(iterations = n)`**
  - 10x interací měření
    - **`Anotace @Measurement(iterations = n)`**
  - 10x iterací benchmarku (spuštění v čistém JRE)
    - **`Anotace @Fork(iterations = n)`**
  - Kromě počtu iterací lze nastavit požadovanou dobu běhu
    - **`@Warmup(iterations = 2, time = 10000, timeUnit = TimeUnit.MILLISECONDS)`**



Děkuji za pozornost

OTÁZKY?