Escape Analysis

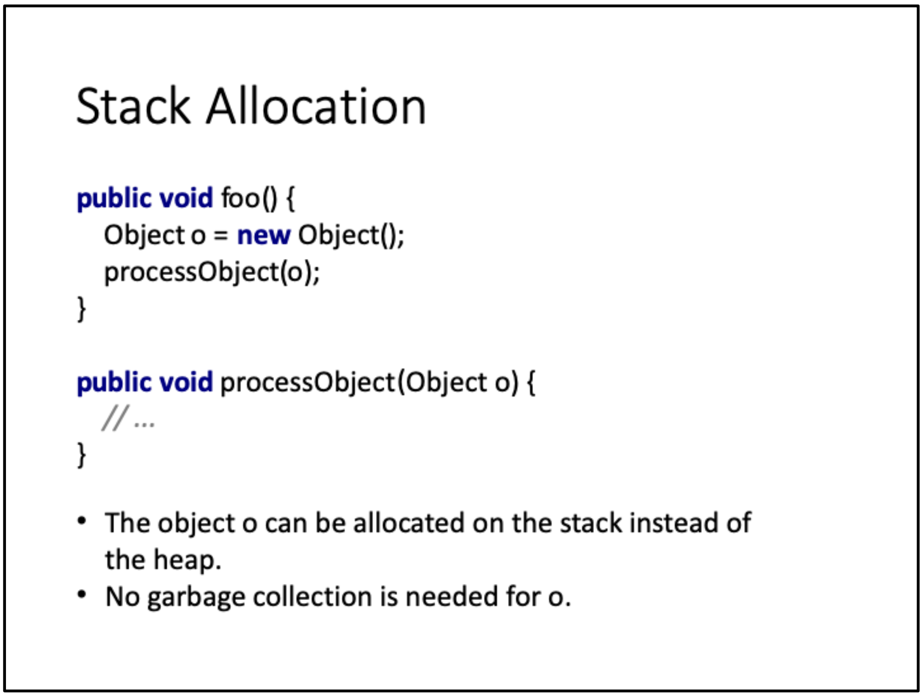
Florian Kübler

What is Escape Analysis?

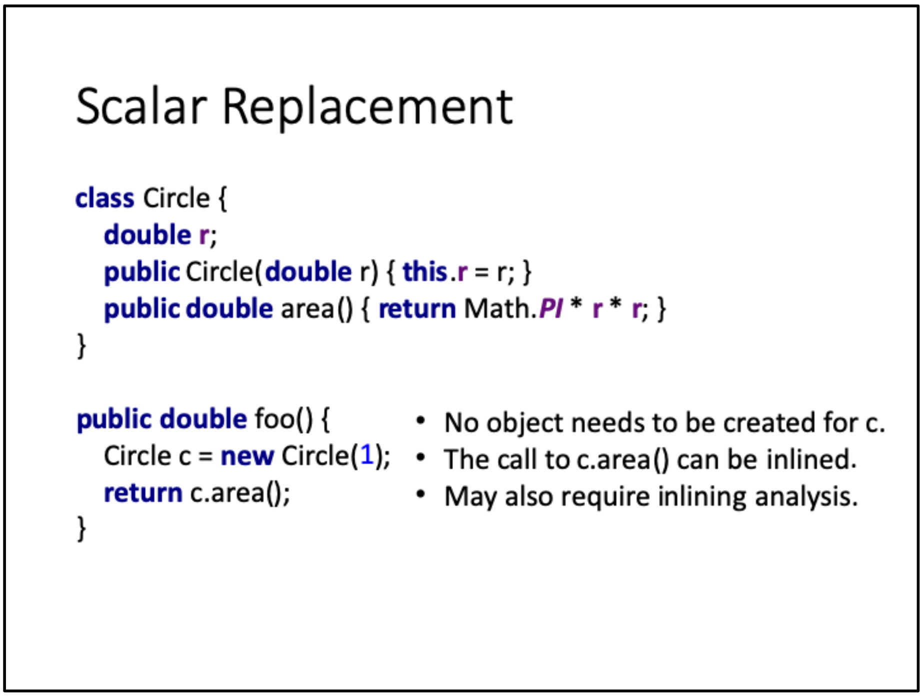
- Escape analysis tries to **statically** reason about the **dynamic scope** of an **object**.

Traditional Use Cases

- Stack-allocations
- Scalar-replacement
- Unnecessary synchronization removal
- Identify pure methods (see purity analysis slides)
- Identify immutable data-structures
- Generate more precise call graphs



Allocating objects on the heap is more expensive than allocating objects on the stack. Furthermore, objects that have been allocated on the stack do not need to be garbage collected, as the stack frame will be release after the method call. Therefore, compilers, JITs and VMs often try to perform stack allocations, (e.g. the JVM does so at least since 9).
Stack allocation is only possible, if after the call of its allocating method, it is not needed anymore.



Some objects do not need to be allocated at all. They only provide some abstraction/modularization and that can be inlined directly.
In the given example, the code of foo could be rewritten/optimized to: return Math.PI * 1 * 1;
This technique is called scalar replacement (performed by the JVM since Java 6).
In order to avoid allocation of objects, either only the allocating object access the object or all methods that access the object can be inlined.
For simplicity, escape analysis often does not perform inlining analysis.

Removing Unnecessary Synchronization

```java
public class AppletViewer {
  // ...
  public URL getCodeBase() {
    Object o = new Object();
    synchronized (o) {
      // ...
    }
    return this.baseURL;
  }
}
```

Synchronization is useless on o, as only this call of getCodeBase has access to it.

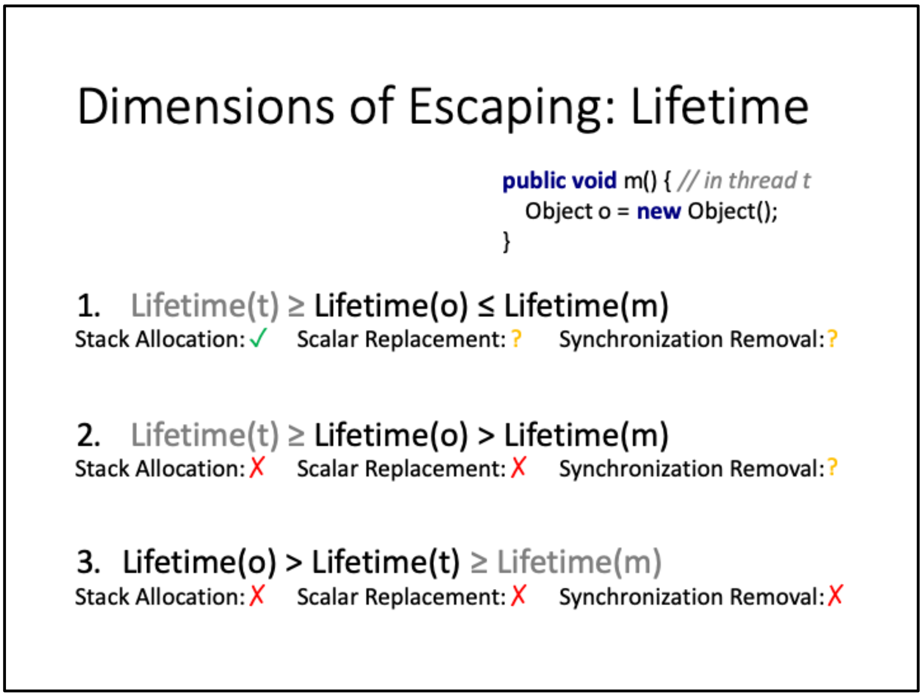
This code was found in JDK 8 update 151

Synchronization instructions introduce a lot overhead within the JVM.
If the object, that is used within the synchronization can not be accessed by any other thread, the synchronization is then useless and may be removed.
The JVM performs this optimization since Java 6.

What is Escape Analysis (cont.)?

- Escape analysis makes a statement about the object’s **lifetime** and **accessibility**
- In contrast to lifetime analysis, escape analysis is more coarse grain and binds the object’s lifetime/accessibility to other entities (in particular **methods** or **threads**)

Accessibility describes, whether other methods or threads may have a reference to it and therefore it is possible to read/write fields of the object ,invoke methods on it, or pass it as an parameter.



We will use *m* for methods and *t* for threads in the rest of the slides.

Dimensions of Escaping: Accessibility

```java
public void m() { // in thread t
    Object o = new Object();
}
```

1. Access(o) = {m, t}

   Stack Allocation: ✓ Scalar Replacement: ✓ Synchronization Removal: ✓

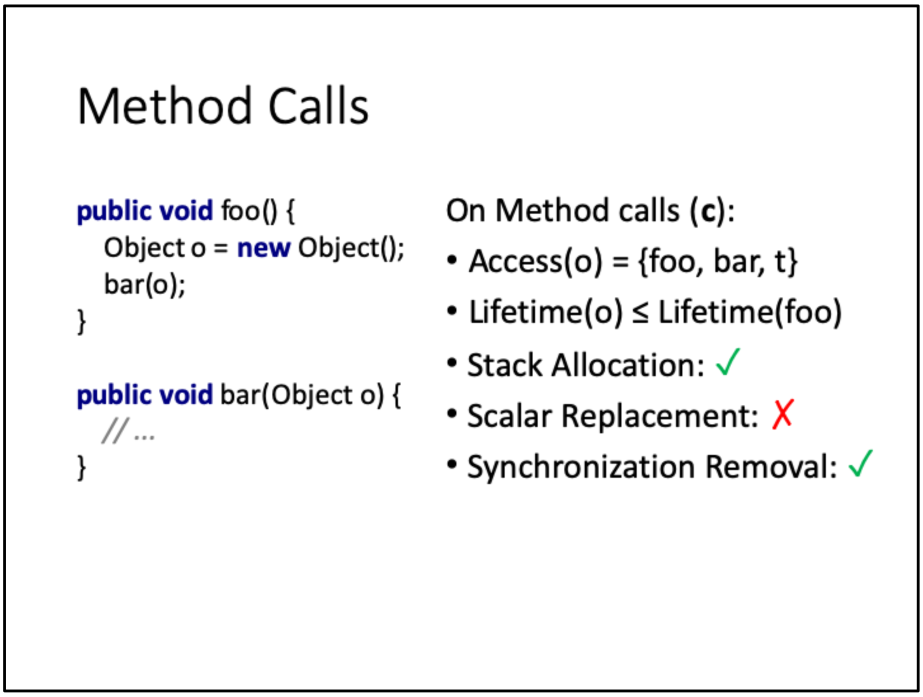
2. $(\exists m' \neq m \,.\, m' \in Access(o)) \wedge (\nexists t \neq t'.t' \in Access(o))$

   Stack Allocation: ? Scalar Replacement: ✗/? Synchronization Removal: ✓

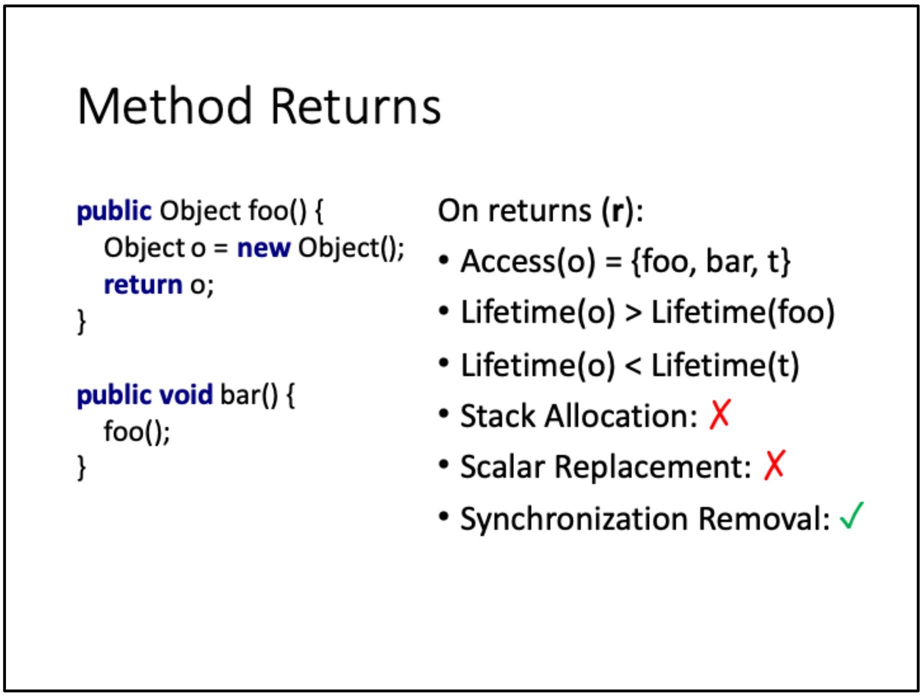
3. $\exists t' \neq t \,.\, t' \in Access(o)$

   Stack Allocation: ✗ Scalar Replacement: ✗ Synchronization Removal: ✗

For case 2, it depends on whether m’ can be inlined



Method calls grant access to the call target.



Access to o will be granted to all callers

Unhandled Exceptions

```java
public void foo() {
   throw new Exception();
}

public void bar() {
   foo();
}
```

On abnormal returns (**a**):

- Access(o) = {foo, bar, t}
- Lifetime(o) > Lifetime(foo)
- Lifetime(o) < Lifetime(t)
- Stack Allocation: ✗
- Scalar Replacement: ✗
- Synchronization Removal: ✓

Mutating the State of Given Parameters

```
public class MyClass {
   Object f;

   public void foo(MyClass p) {
      p.f = new Object();
   }

   public void bar() {
      foo(new MyClass);
   }
}
```

Parameter field writes(**p**):

- Access(o) = {foo, bar, t}
- Lifetime(o) > Lifetime(foo)
- Lifetime(o) < Lifetime(t)
- Stack Allocation: ✗
- Scalar Replacement: ✗
- Synchronization Removal: ?

Writing Static Fields

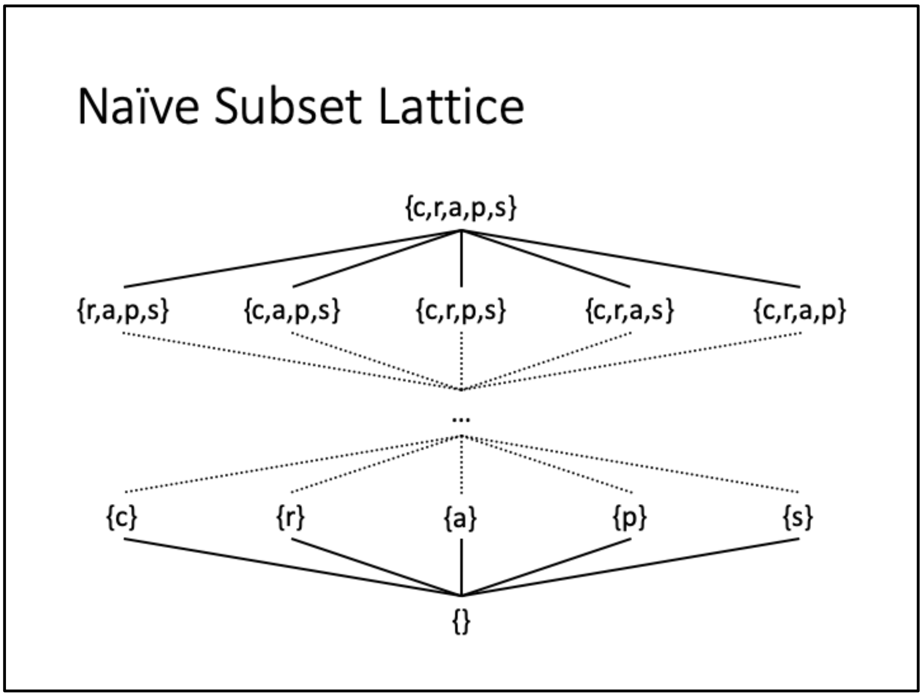
```
public class MyClass {
   public static Object f;

   public void foo() {
      Object o = new Object();
      MyClass.f = o;
   }

   public void bar() {
      foo();
   }
}
```

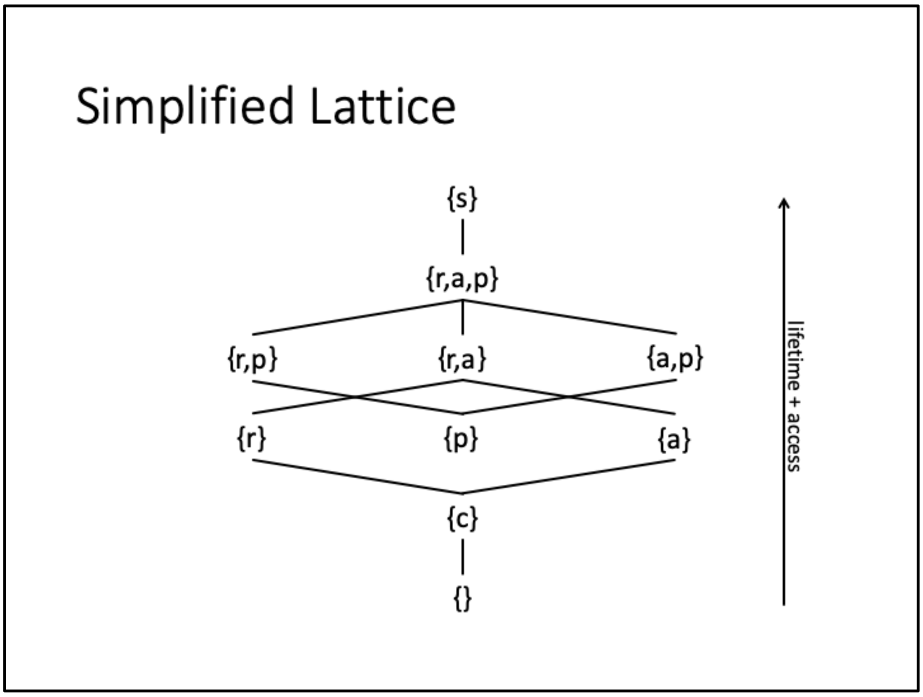
Static field writes(**s**):

- ∀m . m ∈ Access(o)
- ∀t' . t' ∈ Access(o)
- Lifetime(o) > Lifetime(foo)
- Lifetime(o) > Lifetime(t)
- Stack Allocation: ✗
- Scalar Replacement: ✗
- Synchronization Removal: ✗



c: Method call
r: Return value
a: Abnormal return
p: field of parameter
s: static field write

A naïve way to represent escape states would be to model sets of language features that correlate to escape information.
This would result in this subset lattice.



c: Method call
r: Return value
a: Abnormal return
p: field of parameter
s: static field write

The subset lattice has redundant information when mapped back to the impact to lifetime/access properties, e.g. all sets that contain ”s” are a bound for lifetime and access.
We can restrict the lattice to the given one.
Furthermore, it might not be needed to distinguish between “a” and “r”, but this may help clients to distinguish.

Handling Constructor Calls

```
Object o = new Object();
```

```
NEW(java.lang.Object)
DUP
INVOKESPECIAL(java.lang.Object{ void <init>() })
```

- As known from the lecture, the JVM distinguishes between object allocation and invocations of <init>.
- Distinguish between constructor and other calls as <init> does not let escape the object per-se.
- Requires (at least some) inter-procedural analysis if it is not the constructor of java.lang.Object

May-Alias Analysis and Fields

```java
public class MyClass {
   Object f;
   public static Object global;

   public void foo() {
      MyClass c1 = new MyClass();
      MyClass.global = c1; // c1 escapes
      MyClass c2 = c1; // requires may-alias detection
      Object o = new Object();
      c2.f = o; // c2 does escape, therefor o does as well
   }
}
```

Handling fields requires to reason about the aliases and escape information of the base object.

Research Directions

- Use escape information for more precise call graphs for libraries.
- Modularization and soundness of call-graph and points-to analysis
- For lab, seminar or thesis topics as well as for HiWi positions contact me (kuebler@cs.tu-darmstadt.de).

References

- Kotzmann, Thomas, and Hanspeter Mössenböck. "Escape analysis in the context of dynamic compilation and deoptimization." *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*. ACM, 2005.
- Choi, Jong-Deok, et al. "Escape analysis for Java." *Acm Sigplan Notices* 34.10 (1999): 1-19.
- Kotzmann, Thomas, and Hanspeter Mössenböck. "Escape analysis in the context of dynamic compilation and deoptimization." *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*. ACM, 2005.
- https://shipilev.net/jvm/anatomy-quarks/19-lock-elision/
- https://docs.oracle.com/javase/8/docs/technotes/guides/vm/performance-enhancements-7.html
- https://dzone.com/articles/overview-of-javas-escape-analysis
- https://dzone.com/articles/escape-analysis