

I Can Capture You: Optimizing Object-Oriented Programs

IICT 2024

Manas Thakur

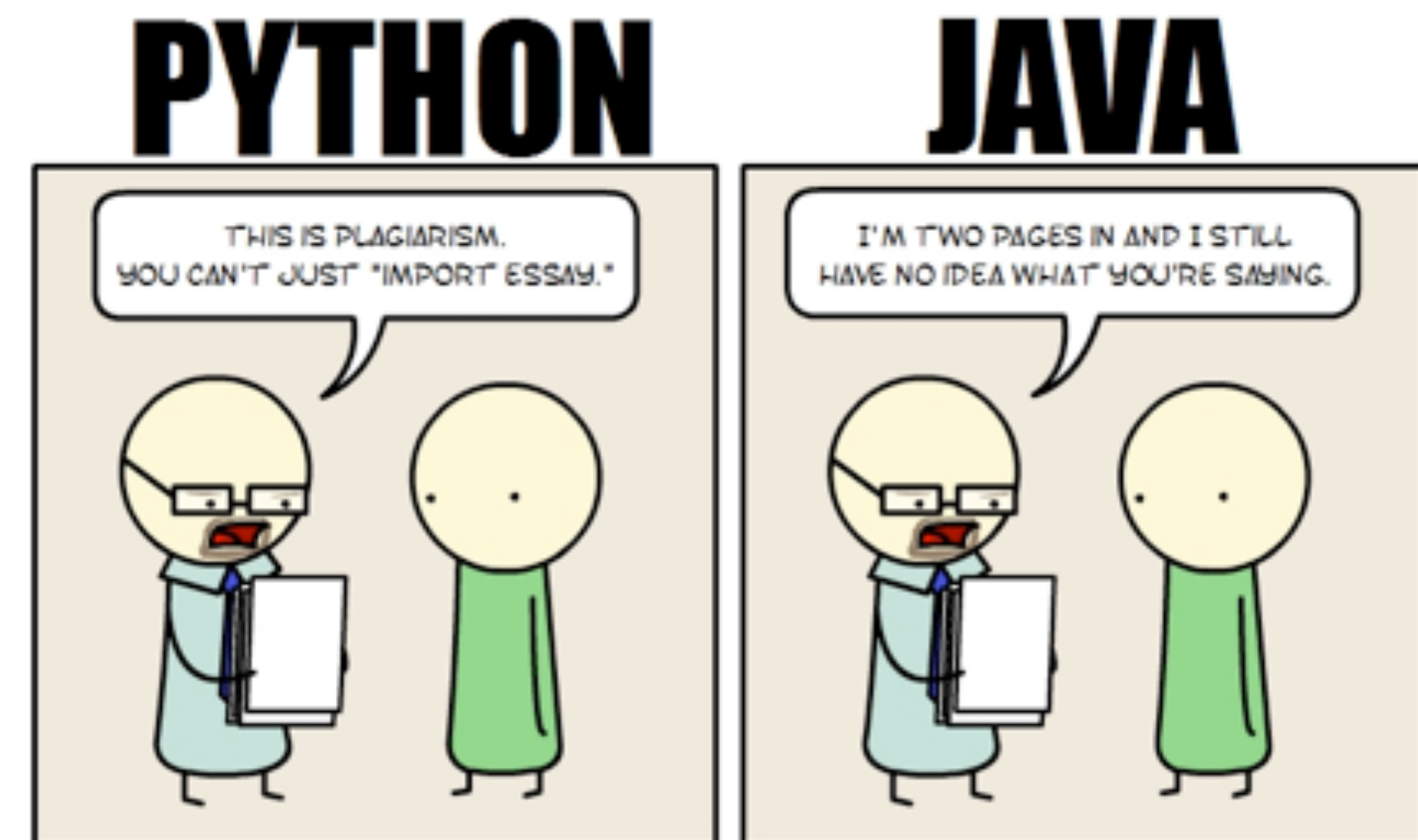


September 29th, 2024

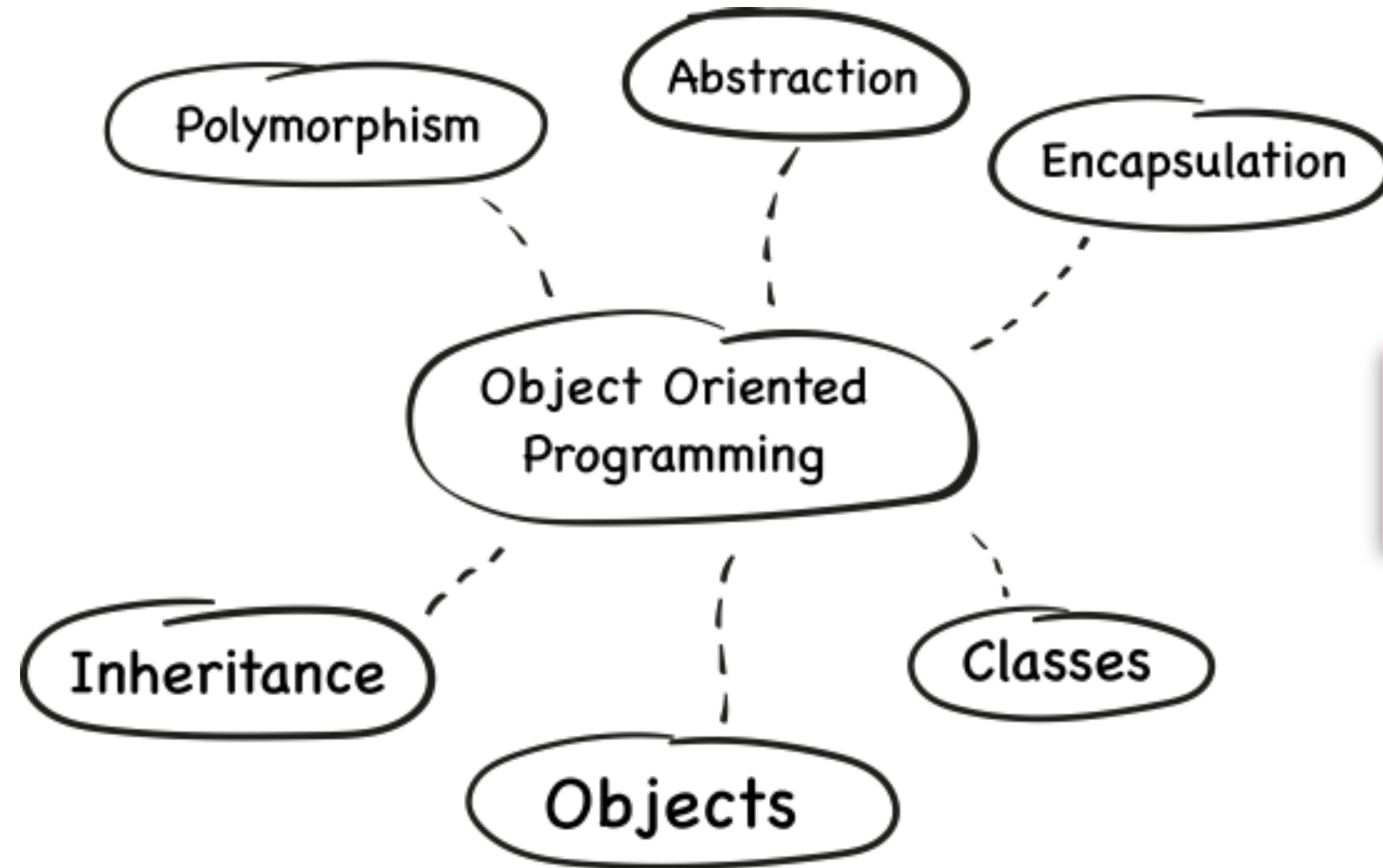
Who cares about OO languages?

Rank	Language	Share
1	Python	28.2 %
2	Java	15.73 %
3	JavaScript	8.91 %
4	C/C++	6.8 %
5	C#	6.67 %
6	R	4.59 %
7	PHP	4.54 %
8	TypeScript	2.92 %
9	Swift	2.77 %
10	Objective-C	2.34 %

Source: PYPL Index, Jan'24.



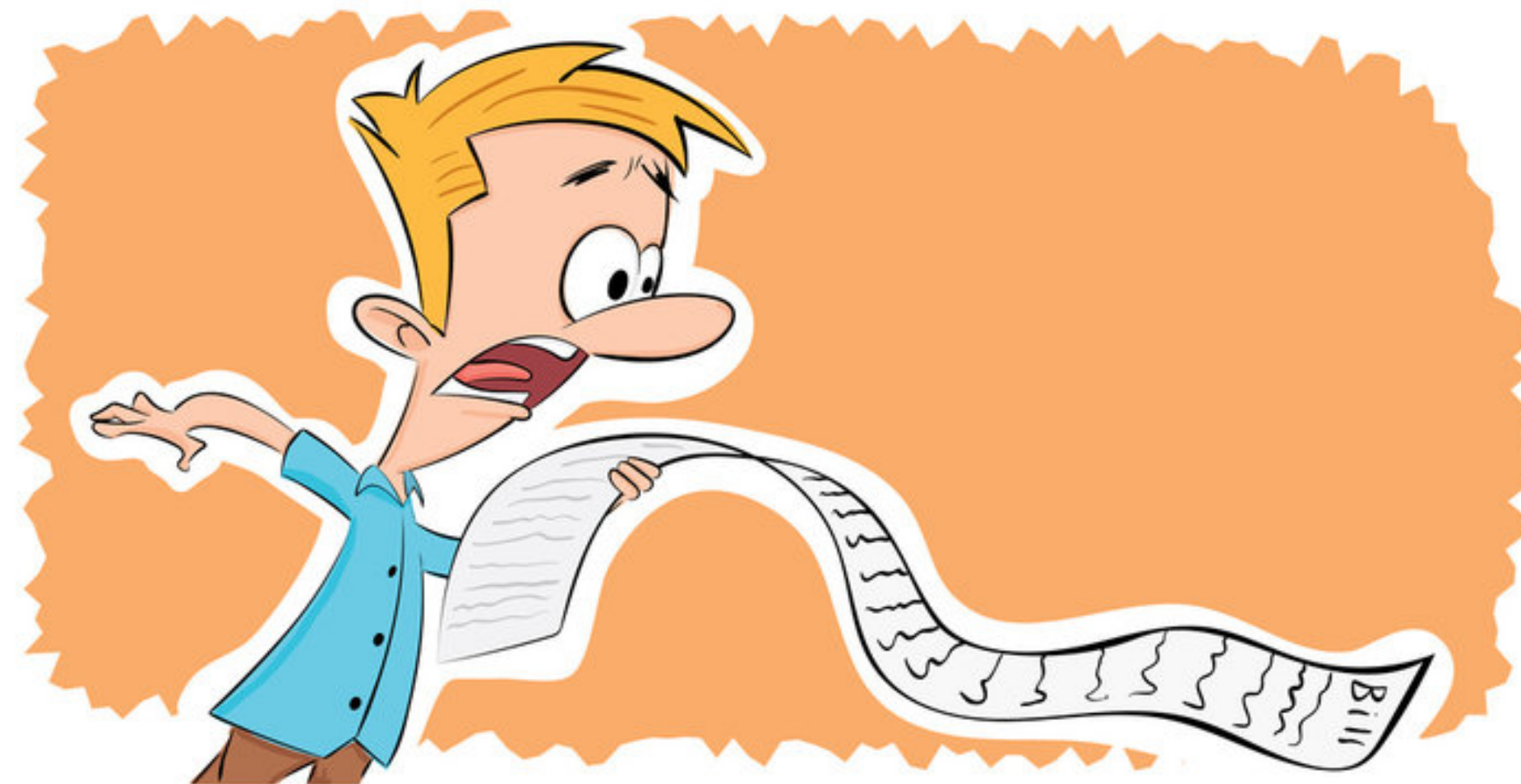
What is special about OO languages?



Managed runtimes additionally offer Garbage Collection.

Let's summarize the costs (say for Java)

- Field access through objects (Abstraction & Encapsulation) involves indirection **Costly**
 - Also adds to the memory footprint **Costly**
- Method overloading (Inheritance & Polymorphism) complicates virtual calls **Costly**
- Type safety (Classes and Objects) requires run-time checks **Costly**
- Collecting garbage requires run-time analysis **Costly**



But OO programs run fast enough!

- OO languages dominate the software-development industry.
- OO code is written for everything ranging from satellites to handheld devices.
- People are even writing virtual machines and kernels using OO languages!



- What (all) brings performance to OO programs?
- Worth learning how the food you eat is cooked!

[Spring24] I taught a COOOL course

- **Fundamentals** (OO abstraction; memory organization).
- **Heap analysis** (points-to information; field- and flow-sensitivity; intra- and inter-procedural analysis; alias analysis; heap cloning).
- **Optimizations involving allocation, field access and deallocation** (pointer and escape analysis; stack allocation; scalar replacement; field privatization; value types and object inlining; object colocation; control-sensitive analysis; garbage collection).
- **Optimizations involving method dispatch** (class hierarchy analysis; rapid type analysis; context-insensitive analysis; object- and type-sensitive analysis).
- **Speculative optimizations** (speculative type resolution; speculative method inlining and polymorphic inline caching; code versioning; deoptimization overheads).
- **Recent developments** (closed-world assumption; dynamic features; mixing AOT and JIT).



Object allocation

- Objects allocated using new in Java are stored on the heap.
- The only way of allocating objects in Java is using new.

Java code:

```
x = new ArrayList<T>();
```

Object layout:

OFF	SZ	TYPE	DESCRIPTION
0	8		(object header: mark)
8	4		(object header: class)
12	4	int	AbstractList.modCount
16	4	int	ArrayList.size
20	4	java.lang.Object[]	ArrayList.elementData
Instance size: 24 bytes			

Bytecode:

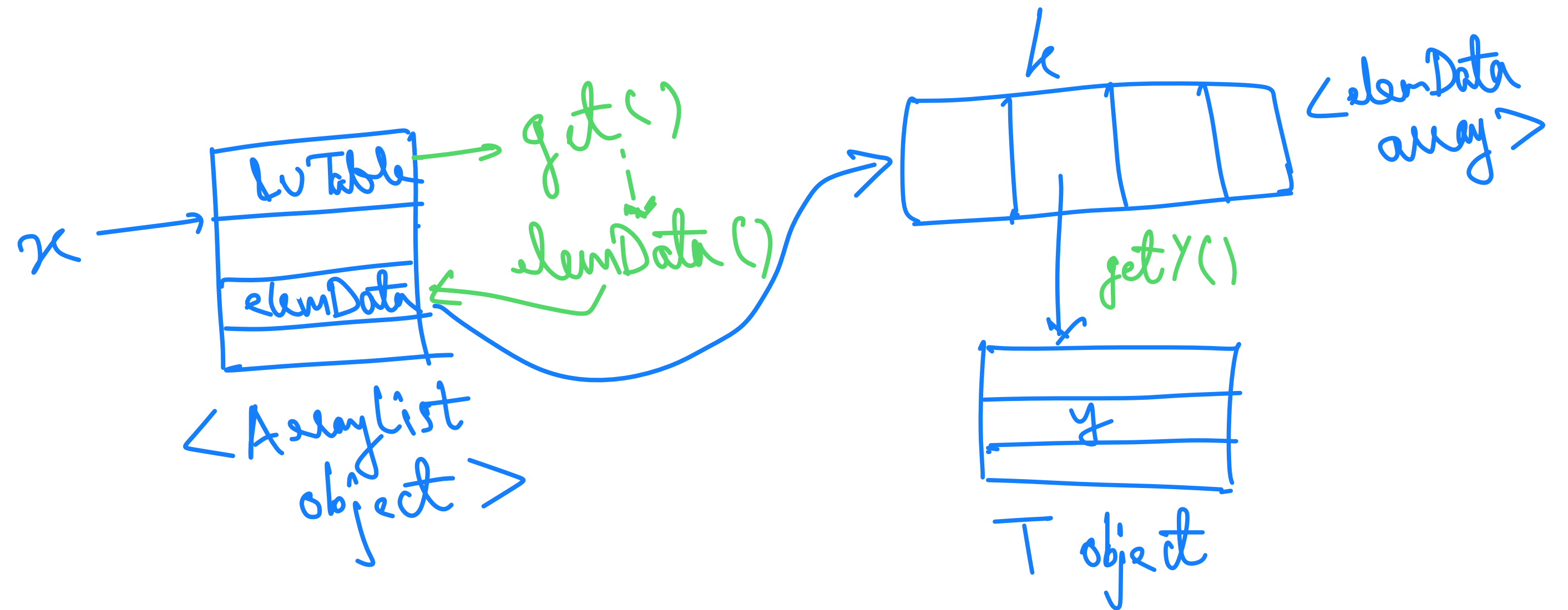
```
0: new          #7          // class java/util/ArrayList
3: dup
4: invokespecial #9          // Method java/util/ArrayList."<init>":()V
7: astore_1
```



Field access

- Each field access requires a memory load.
- Field accesses are often wrapped inside *getter* calls.
- Reference fields may load objects from different parts of the heap, not necessarily resident in the same cache block.

```
class T { int y; ...}  
...  
x = new ArrayList<T>();  
...  
int y = ((T) x.get(k)).getY();
```



- Three memory loads (cache misses) if different objects are on different pages (blocks).

Garbage collection

```
a = new T(); // O1
a.f = new T(); // O2
foo(a);
a.f = b;
// Can we now collect O1 and/or O2?
```

- If an object is not reachable from any reference any more, then yes.
- Even if it is reachable but the reference is not used, then yes.
- Knowing this **requires**:
 - Maintaining reachability at run-time.
 - Liveness analysis.
 - Interprocedural analysis (because foo may create newer references to O₁ and/or O₂).
- Even concurrent GCs snatch resources from the program.

Lesser the work the GC has to do, the better.



Can ~~we~~ the compiler get rid of objects?

- Replace an object by scalar variables representing its constituents, and use them instead of its fields:

```
class T {  
    int f1;  
    int f2;  
    public T() {  
        f1 = 10;  
        f2 = 20;  
    }  
}
```

```
T x = new T();  
...  
z = x.f1 + x.f2;
```



```
int x_f1, x_f2;  
x_f1 = 10;  
x_f2 = 20;  
...  
z = x_f1 + x_f2;
```

- An optimization called **SCALAR REPLACEMENT**.
- Gets rid of object allocations altogether, and just adds local variables on stack.
- Constant offsets from stack pointer, better cache locality, no garbage collection.



When can we not scalar-replace an object?

- What if we actually *needed* the object?

```
T x = new T();  
...  
foo(x);
```

```
T x = new T();  
...  
if (x == y) {...}
```

- What if somebody else could modify the object's fields?

```
static T global;  
...  
T x = new T();  
global = x;
```

- Scalar replacement is good, but **several *highly used* Java constructs disallow object decomposition.**

Thomas Kotzmann and Hanspeter Mössenböck. “Escape Analysis in the Context of Dynamic Compilation and Deoptimization”. VEE 2005.



Can we scalar-replace an object in *some* paths?

- Many objects cannot be scalarized only in *cold* code branches. Scalarize them in the beginning, and *rematerialize* them when needed:

```
T x = new T(10, 20);  
if (*) {  
    foo(x);  
}  
// Local field accesses
```



```
int x_f1 = 10;  
int x_f2 = 20;  
if (*) {  
    foo(new T(10, 20));  
}  
// Local field accesses
```

- Rematerialization **requires maintaining a run-time map** from scalarized objects to the values of their fields.

Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. “Partial Escape Analysis and Scalar Replacement for Java”. CGO 2014.



Can we scalar-replace an object *temporarily*?

- If the object has not *escaped yet*, scalarize (or *privatize*) its field(s) locally where they are accessed frequently:

```
T x = new T();  
while (*) {  
    z = x.f;  
    ...  
    x.f = y;  
}  
global = x;
```



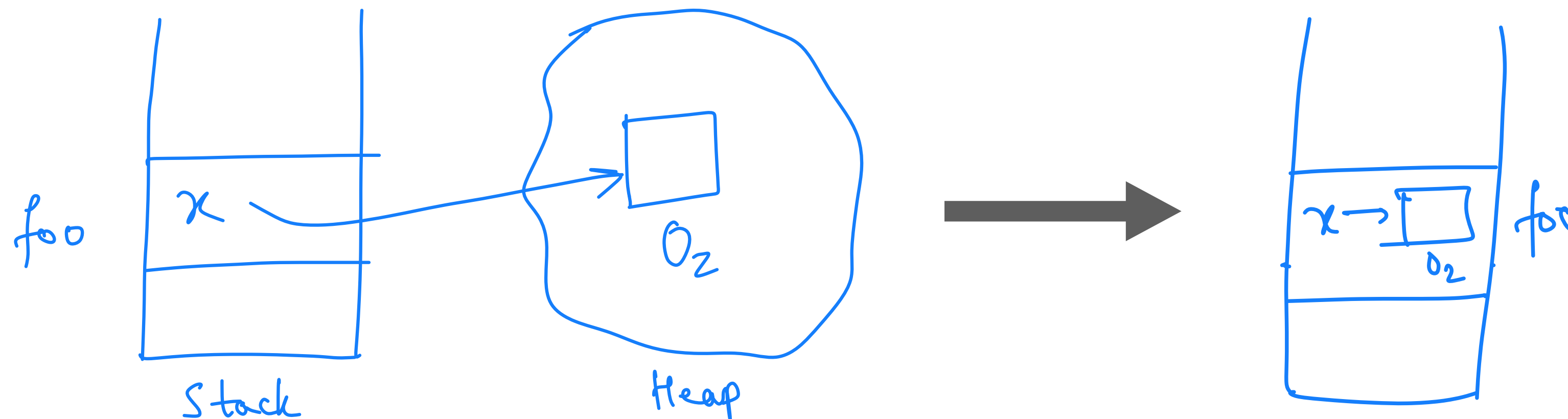
```
int t = x.f;  
while (*) {  
    z = t;  
    ...  
    t = y;  
}  
x.f = t;  
global = x;
```

- An optimization called **FIELD PRIVATIZATION**.
- Gets rid of objects inside loops.
- A smarter cousin: **SPLIT SCALARIZATION**.

Vijay Sundaresan, Daryl Maier, Krishna Nandivada Venkata, and Manas Thakur. “Split-Scalarization of Thread-Local Objects in Optimized Object Code”. (Filed) US Patent, 2023.

Can we allocate objects on the stack itself?

- Allocate an object on the stack of its allocating method instead of on the heap:



- An optimization called **STACK ALLOCATION**.
- Object components still allocated *contiguously*, and hence available *as an object*.
- Constant offsets from stack pointer, better cache locality, free garbage collection.

When can we not allocate an object on stack?

- Objects that outlive their allocating method cannot be allocated on the stack-frame of that method:

```
1: class C {
2:     static D g;
3:     void foo() {
4:         D x = new D(); //O4
5:         D y = new D(); //O5
6:         x.f = new D(); //O6
7:         bar(x);
8:         ...
9:     }
10:    void bar(D p) {
11:        g = p.f;
12:    }
13: }
14: class D { D f; }
```

- O₄ and O₅ are not accessible beyond the life-time of foo, and hence are stack allocatable.
- O₆ (and any objects reachable from O₆) may outlive foo, and hence aren't stack allocatable.
- Requires performing *points-to analysis* across procedure boundaries, and then checking reachability.
- Popularly called **ESCAPE ANALYSIS**.



Escape analysis

- Tries to classify the (abstract) objects in a program into one of 3 states:
 - DoesNotEscape Can be scalar replaced.
 - MethodEscape Can be stack allocated.
 - GlobalEscape Need to remain on heap.
- Known to be one of the most impactful analyses in terms of improving the performance of OO programs.
- In managed runtimes (e.g. JVMs and .NET), performed typically by their JIT compilers.
- Suffers from the imprecision of resource-constrained JIT analyses.



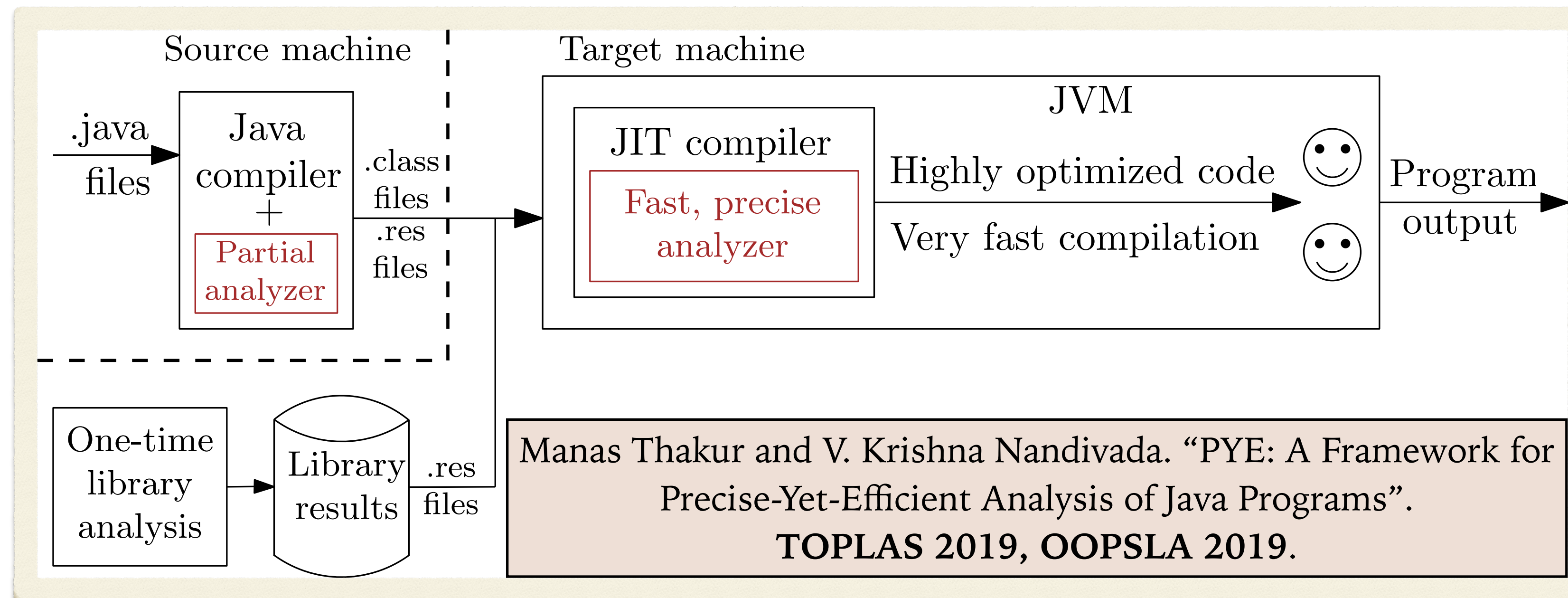
Escape analysis in JIT compilers

- Typical JITs perform complex program analyses only at higher optimization levels (*hot* methods).
- Certainly no budget for interprocedural analysis. (Speculative profile-based method inlining gives some improvements.)
- e.g., the JIT compiler of OpenJ9 can stack allocate O_5 only at hot+ levels of compilation, and O_4 only when the runtime is able to inline (or peek into) `bar`.
- Static analysis, on the other hand, is extremely conservative in presence of calls to library methods.

```
1: class C {
2:     static D g;
3:     void foo() {
4:         D x = new D(); //O4
5:         D y = new D(); //O5
6:         x.f = new D(); //O6
7:         bar(x);
8:         ...
9:     }
10: void bar(D p) {
11:     g = p.f;
12: }
13: class D { D f; }
```



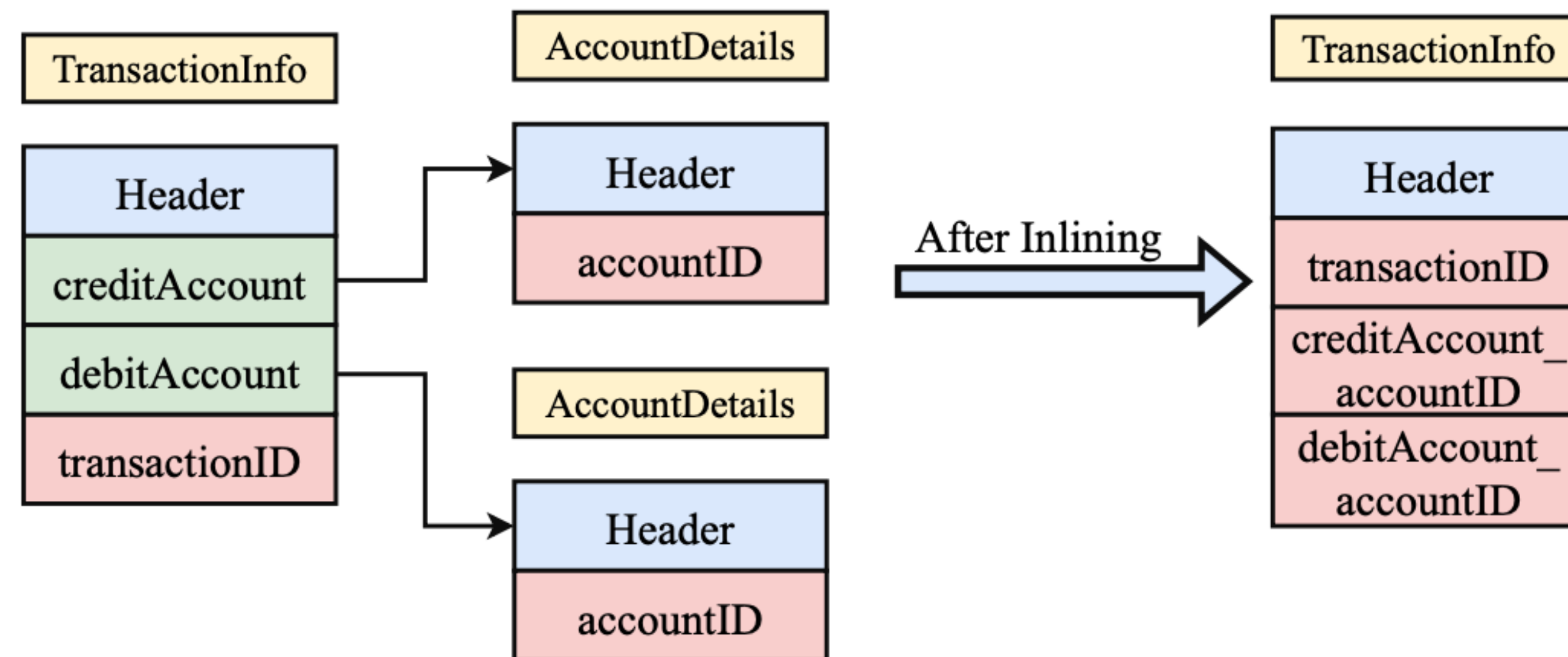
PEA in PYE



- Analyze applications and libraries independently, while remembering the dependencies.
- Resolve the dependencies during JIT compilation.
- Advantage: Results of precise analyses can be obtained to enable sophisticated optimizations in the JVM, without much JIT cost.

Is the story over for non-stack-allocatable objects?

- An object that is often accessed through a field of a “container”, can be allocated inside the container:



- An optimization called **OBJECT INLINING**.
- Reduces memory footprint, field indirections, improves cache locality.

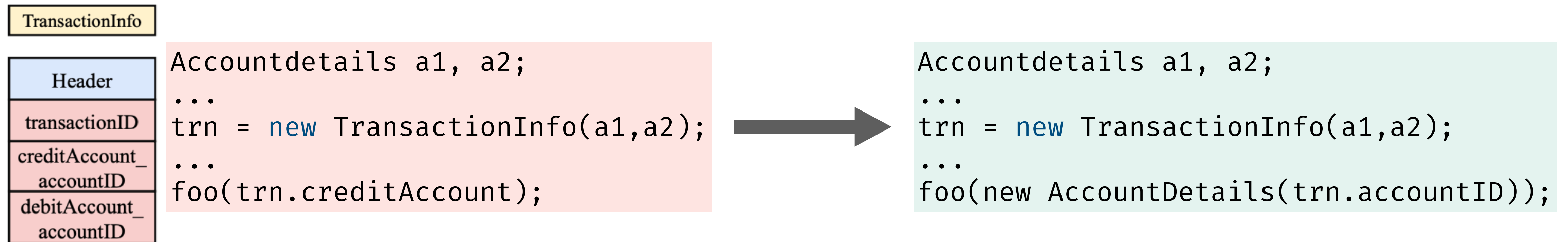
When can we (not) inline an object?

- When there are mutable references to the object from outside the container in which it is inlined.
- Unless we update those references to point inside the container == > **complicated and risky** (may have to tweak with accessibility of the container, as well as disable the possibility of its stack allocation!).
- Java proposes inlining objects of **immutable value types** (an upcoming feature)
 - All objects of value-type classes would be inlined inside all their containers, subject to a user-defined size threshold, during JIT compilation.



When *should* we not inline an object?

- When an inlined value-type object is needed in its entirety, it has to be *recreated*:



- Use *escape analysis* and identify inlined objects that need to be recreated more often than being accessed through container fields!

Arjun Harikumar, Lorenzo Prosch, and Manas Thakur. **WIP.**



So sometimes we won't inline an object, but...

- The essential problem with inlining objects is that it destroys their header.
- The benefits included reduced memory, fewer indirections, and better cache locality.
- *But we can still achieve the last two without destroying the header!*
- We can allocate a value-type object near (or next to) its container.
- An optimization called **OBJECT COLOCATION**.



Rodrigo Bruno, Vojin Jovanovic, Christian Wimmer, and Gustavo Alonso.
“Compiler-Assisted Object Inlining with Value Fields”. PLDI 2021.



We won't rest yet

- Can we aggressively allocate objects on stack based on a precise static escape analysis and move them to the heap, if needed (**WHEN?**), dynamically?

More to come in the afternoon!



Aditya Anand, Solai Adithya, Swapnil Rustagi, Priyam Seth,
Vijay Sundaresan, Daryl Maier, V. Krishna Nandivada, and Manas Thakur.
“Optimistic Stack Allocation and Dynamic Heapification for Managed Runtimes”.
PLDI 2024.

			
Organization	Attending	Schedule	New
9:00 - 10:00	Invited talk Optimizations for Object Oriented Programs Manas Thakur		
10:00 - 10:20	CoS-SSA: SSA for Context-Sensitive Interprocedural Analysis Pritam Gharat, Uday P. Khedker, Alan Mycroft, Supriya Bhide and Aditya Pradhan		
10:20 - 10:40	A Correspondence Between ϕ-function Placement in SSA and Reaching Definitions Analysis Supriya Bhide, Uday Khedker and Pritam Gharat		
10:40 - 11:00	pliron: An Extensible IR Framework in Rust Vaivaswatha Nagaraj		
11:00 - 11:30	Tea break		
11:30-11:50	SLIM: A High-Level Abstraction on LLVM IR Suitable for Program Analysis Aditi Raste, Aditya Pradhan, Akshat Oke and Uday Khedker		
11:50 - 12:10	Why generating Three Address Code for Javascript is hard Meetesh Kalpesh Mehta		
12:10-12:30	Program Analysis for Managed Runtimes in Presence of Dynamic Features Aditya Anand		
12:30-12:50	Engineering behind OCaml's Effect handlers Manas Jayanth		



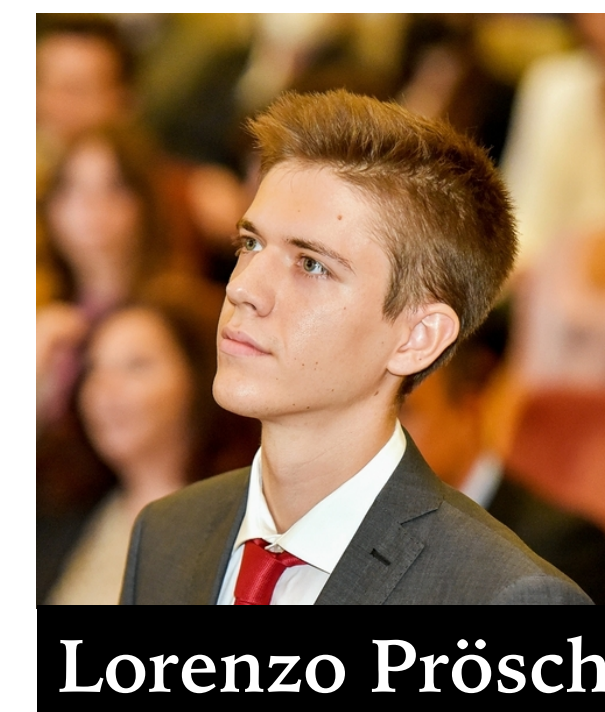
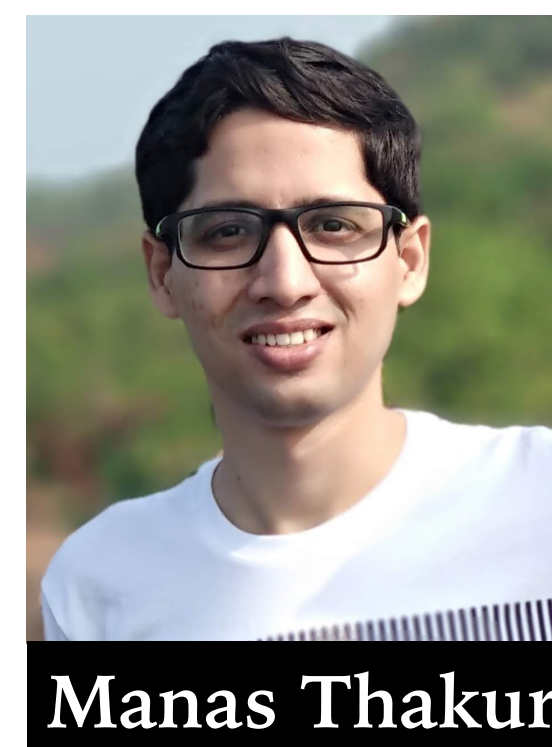
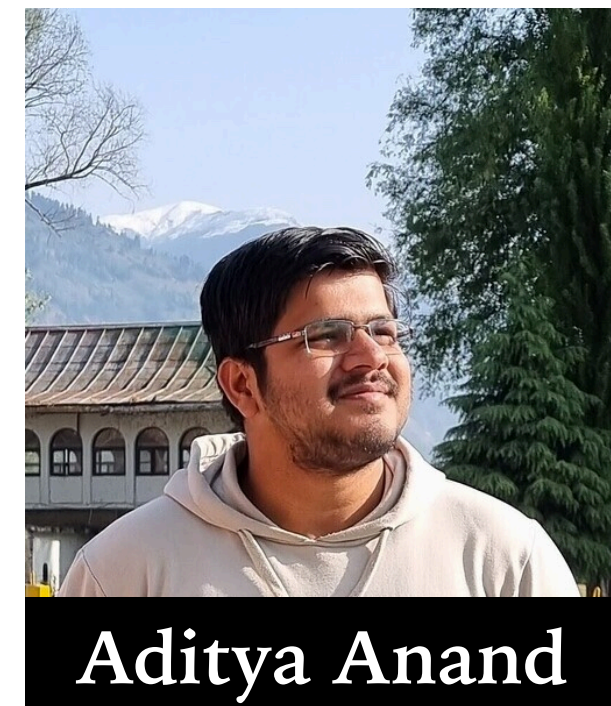
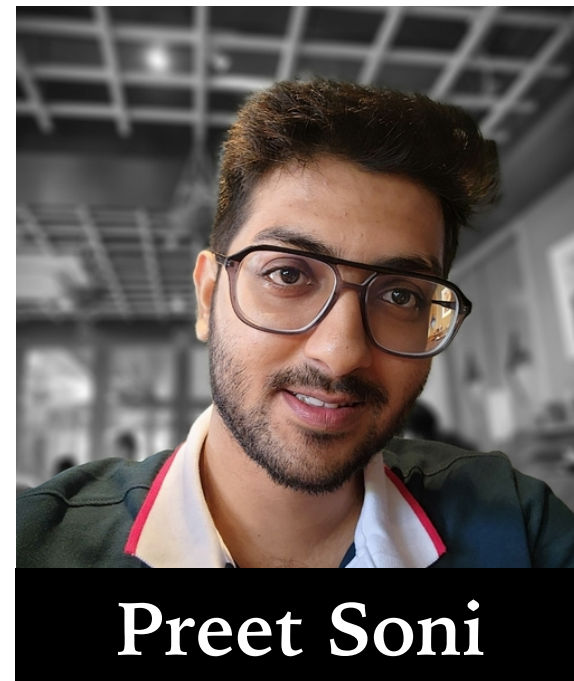
And not even after that...

- Can we allocate objects on caller's stack when they cannot be allocated on the allocating method's stack?
- Can we refactor programs to enable more scalar replacement and more object inlining?
- Can we improve *partially computed* static-analysis results with speculation performed during JIT compilation?
- Can we optimize across language boundaries in polyglot programs?
- Can we use static (offline) analysis to improve JIT decisions in dynamically typed languages such as R and JavaScript?

Meetesh Mehta, Sebastian Krynski, Hugo Gualandi, Manas Thakur, and Jan Vitek.
“Reusing Just-In-Time Compiled Code”. OOPSLA 2023.



My torch bearers at IICT



CompL@CSEIITB (+more)