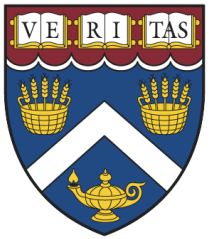
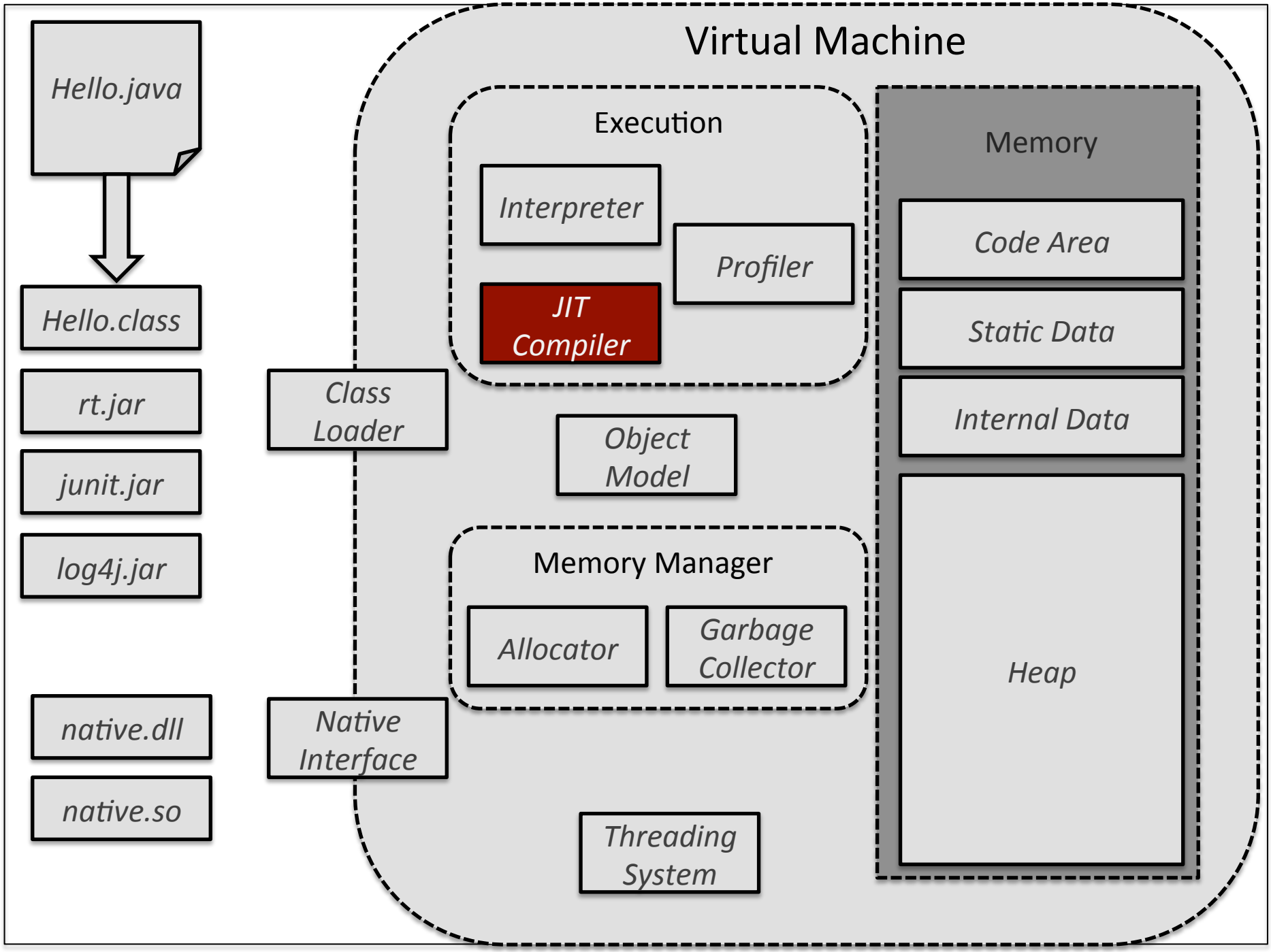
JIT Compilation




Hello.java
Hello.class
rt.jar
junit.jar
log4j.jar
native.dll
native.so
Virtual Machine
Execution
Interpreter
Profiler
JIT Compiler
Class Loader
Object Model
Memory Manager
Allocator
Garbage Collector
Native Interface
Threading System
Memory
Code Area
Static Data
Internal Data
Heap

Jit Compilers

- Highly optimizing compilers
- Can't just use existing static compilers
  - Need to track references for GC
  - Memory model semantics
  - Time constraints (pause times)

Bytecode Compilers

- Remember that javac compiler is very basic
  - Only straightforward optimizations
- VM is responsible for application performance
  - One place for optimization
  - Compiler can be sophisticated
- Other compiler writers don't have to optimize

Compiler Details

- Examples will be in Java code
  - We're interested in concepts, not details
  - Assembly code isn't a prerequisite
- We're interested in optimizing for dynamic code
  - Some optimizations are standard
  - Others are VM-specific
- Compiler design is a full semester's material
  - Take CSCI E-95 for a bigger picture

Instruction Caching

- We talked about data caching last week
- Instructions are read from memory as well
  - Most modern processors have a split L1 cache
  - L2 cache and higher are merged
- Instructions and data have different locality
- Think about I-Cache performance when optimizing

Optimizing Compilation

- Transforming a program to run faster
- Must maintain the semantics of the original
  - Optimizations must be transparent to developers
- Optimizers are not bug-compatible
  - Program errors may show up at higher optimizations
  - More often a problem in native code

Profile-Driven Optimization

- Some optimizations are expensive
  - Involve major code restructuring
  - Require computation over large chunks of code
- We've seen how the VM can help
  - Identify important parts of the code
  - Focus compilation time for best results
- Optimization difficult in a dynamic environment

Optimistic Compilers

- Start with profiling information
- Assume that it's correct
  - Sometimes based on current program state
  - Sometimes guesses based on past behavior
- Need an escape hatch if it's wrong
  - Slow path for bad predictions
- Performance rests on assumptions being correct

Compilation Over Time

- Trade-off for early compilation
  - Compilation increases startup time
  - Benefit to compiling early
- Can run compilation in the background
  - Little slow-down for single-threaded applications
- Compilation overhead decreases over time
  - Long application run, less time as a percentage

Compilation Phases

- Compiler runs in multiple passes
- Each phase produces a correct program
  - No inter-phase dependencies
- Phases can build on each other
  - Output of one phase makes the next more effective

Constant Folding

- Constant values are guaranteed not to change
  - `static final int CONSTANT = 10;`
- We can embed the values directly in the code
  - Save on a `GETSTATIC` call
- Variables can be determined to be constant
  - Check method code for assignments

```
public static final int N = 7;

public int computeValue() {
    int value = N;
    int square = value * value;
    return square - value;
}
```

```
public static final int N = 7;

public int computeValue() {
    int value = N;
    int square = value * value;
    return square - value;
}
```

```
public static final int N = 7;

public int computeValue() {
    int value = 7;
    int square = value * value;
    return square - value;
}
```

```
public static final int N = 7;

public int computeValue() {
    int value = 7;
    int square = value * value;
    return square - value;
}
```

```
public static final int N = 7;

public int computeValue() {
    int value = 7;
    int square =    7  *    7   ;
    return square - value;
}
```

```
public static final int N = 7;

public int computeValue() {
    int value = 7;
    int square =    7  *    7   ;
    return square - value;
}
```

```
public static final int N = 7;

public int computeValue() {
    int value = 7;
    int square =    7  *    7   ;
    return square -    7  ;
}
```

```
public static final int N = 7;

public int computeValue() {
    int value = 7;
    int square =    7  *    7   ;
    return square -    7  ;
}
```

```
public static final int N = 7;

public int computeValue() {
    int value = 7;
    int square =      49      ;
    return square -   7   ;
}
```

```
public static final int N = 7;

public int computeValue() {
    int value = 7;
    int square =        49      ;
    return square -     7   ;
}
```

```
public static final int N = 7;

public int computeValue() {
    int value = 7;
    int square =        49      ;
    return    49    -     7   ;
}
```

```
public static final int N = 7;

public int computeValue() {
    int value = 7;
    int square =        49      ;
    return    49    -    7   ;
}
```

```
public static final int N = 7;

public int computeValue() {
   return   49   -   7  ;
}
```

```
public static final int N = 7;

public int computeValue() {
    return  49  -  7 ;
}
```

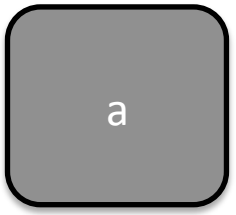
```
public static final int N = 7;

public int computeValue() {
    return 42;
}
```

Escape Analysis

- Variables in Java are scoped
  - Class-level fields can be public or private
  - Local variable scope defined by braces
- References can escape their scopes
  - Passed to external functions
  - Stored in heap objects
- Non-escaping values can be optimized

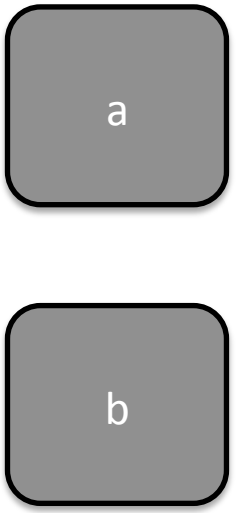
```
class ClassA {
    public static void main(final String[] args) {
        final ClassA a = new ClassA();
        final ClassB b = new ClassB(a);
    }
}

class ClassB {
    private final ClassA a;

    public ClassB(final ClassA a) {
        this.a = a;
    }
}
```

```
class ClassA {
    public static void main(final String[] args) {
        final ClassA a = new ClassA();
        final ClassB b = new ClassB(a);
    }
}

class ClassB {
    private final ClassA a;

    public ClassB(final ClassA a) {
        this.a = a;
    }
}
```

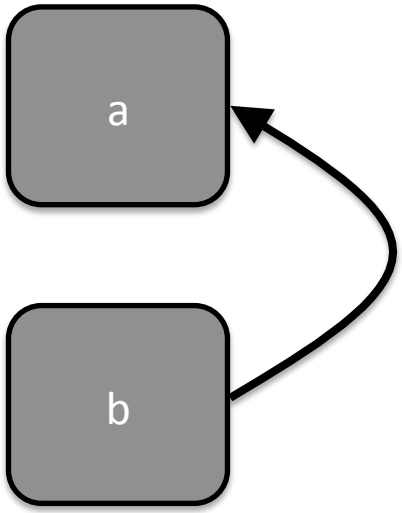



```
class ClassA {
    public static void main(final String[] args) {
        final ClassA a = new ClassA();
        final ClassB b = new ClassB(a);
    }
}

class ClassB {
    private final ClassA a;

    public ClassB(final ClassA a) {
        this.a = a;
    }
}
```

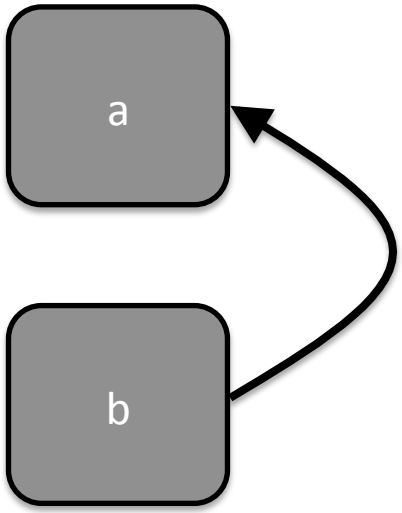
```
class ClassA {
    public static void main(final String[] args) {
        final ClassA a = new ClassA();
        final ClassB b = new ClassB(a);
    }
}

class ClassB {
    private final ClassA a;

    public ClassB(final ClassA a) {
        this.a = a;
    }
}
```

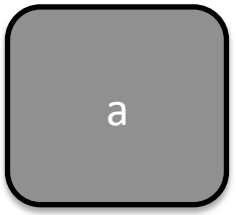


```
class ClassA {
    public static void main(final String[] args) {
        final ClassA a = new ClassA();
        final ClassB b = new ClassB(a);
    }
}

class ClassB {
    private final ClassA a;

    public ClassB(final ClassA a) {
        this.a = a;
    }
}
```

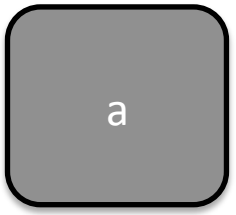


```
class ClassA {
    public static void main(final String[] args) {
        final ClassA a = new ClassA();
        final ClassB b = new ClassB(a);
    }
}

class ClassB {
    private final ClassA a;

    public ClassB(final ClassA a) {
        this.a = a;
    }
}
```






```
class ClassA {
    public static void main(final String[] args) {
        final ClassA a = new ClassA();
        final ClassB b = new ClassB(a);
    }
}

class ClassB {
    private final ClassA a;

    public ClassB(final ClassA a) {
        this.a = a;
    }
}
```




```
class ClassA {
    public static void main(final String[] args) {
        final ClassA a = new ClassA();
        final ClassB b = new ClassB(a);
    }
}

class ClassB {
    private final ClassA a;

    public ClassB(final ClassA a) {
        this.a = a;
    }
}
```

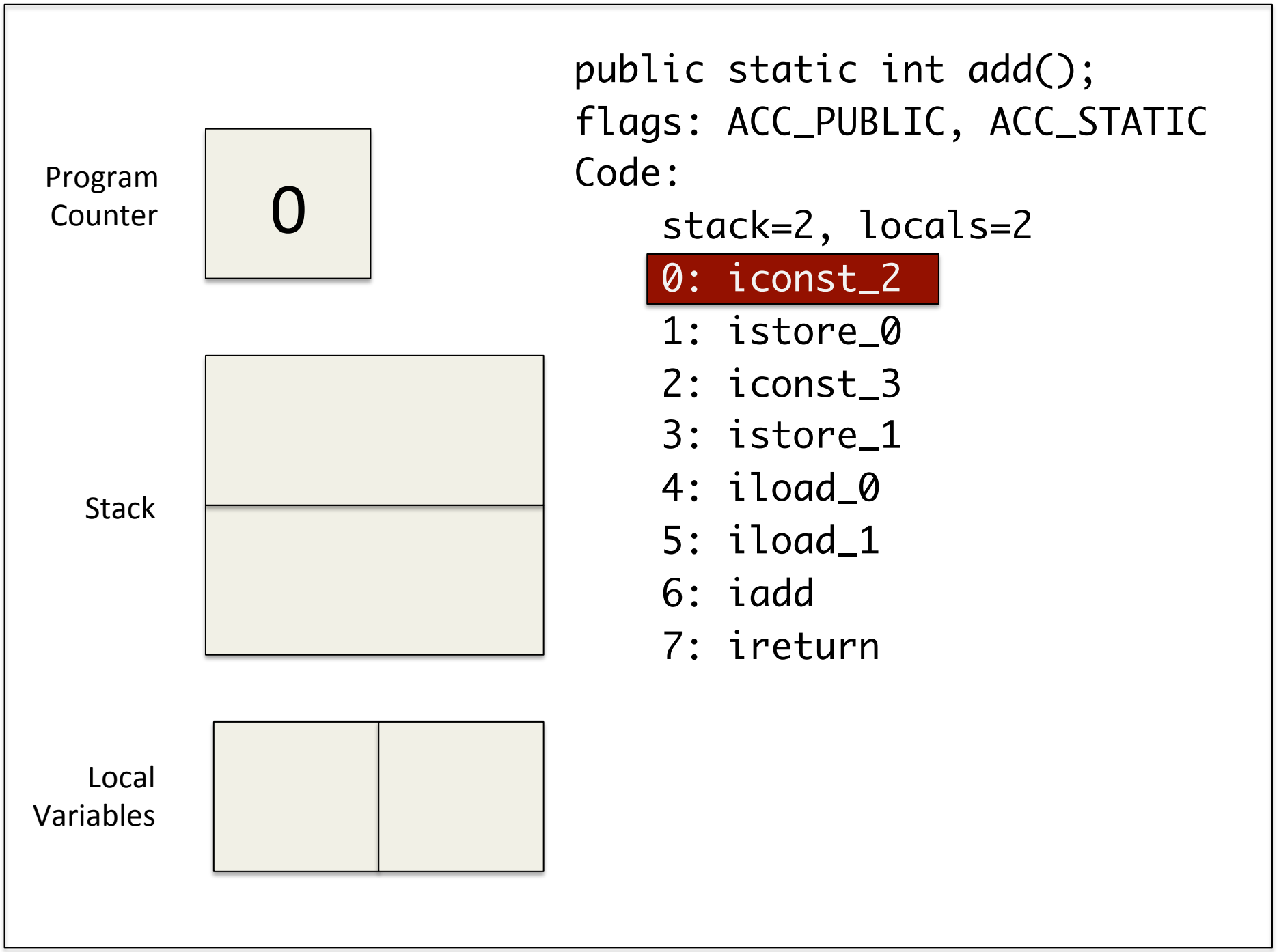
```
class ClassA {
    public static void main(final String[] args) {
        final ClassA a = new ClassA();
        final ClassB b = new ClassB(a);
    }
}

class ClassB {
    private final ClassA a;

    public ClassB(final ClassA a) {
        this.a = a;
    }
}
```

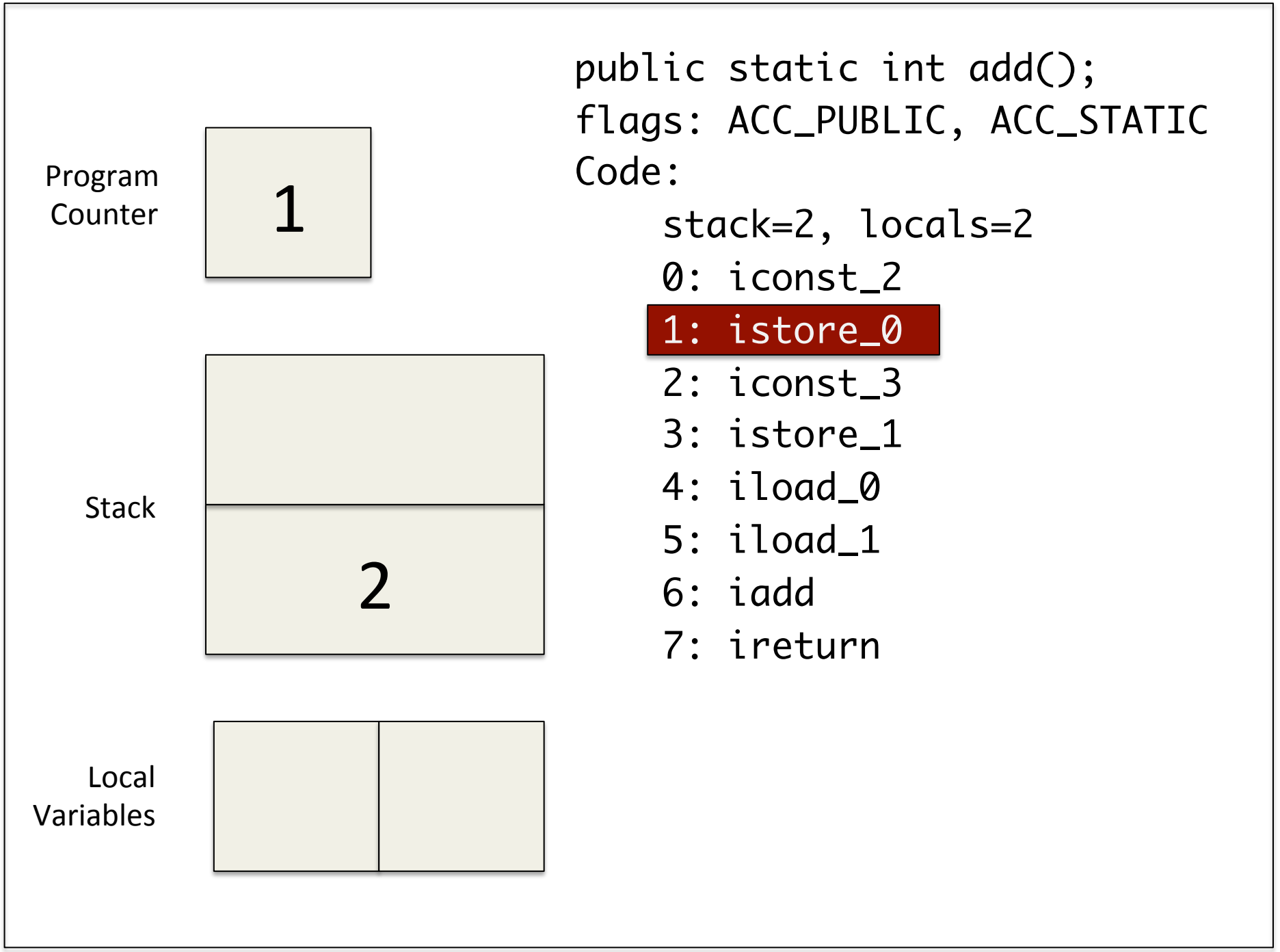
Loads and Stores

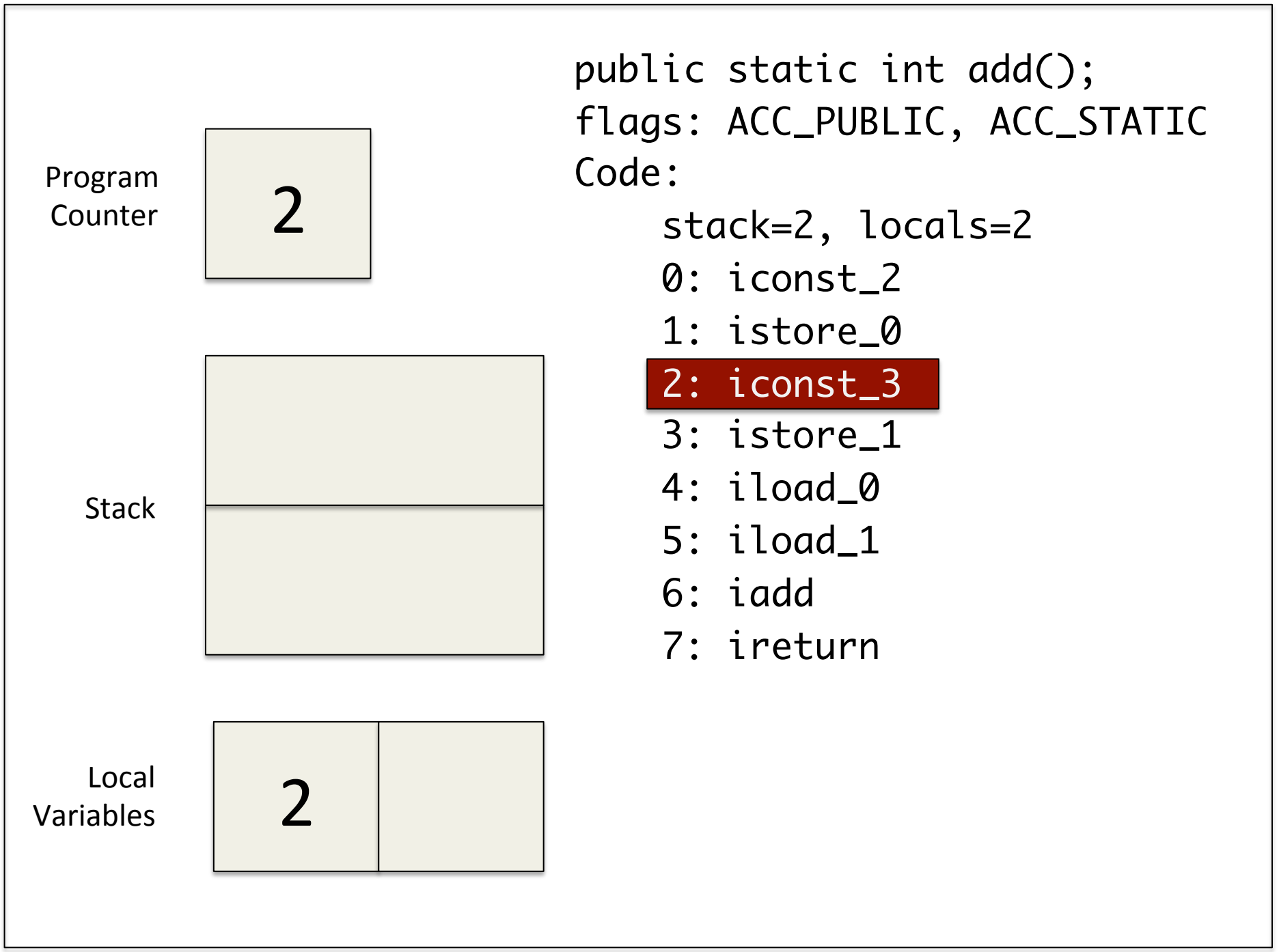
- We've seen a lot of redundant loads and stores
  - Push a value to the stack
  - Store it in a local variable
  - Read it again to pass to a method
- If the local variable isn't read, no need to store it
  - Scan over the method's code
  - Check for load instructions

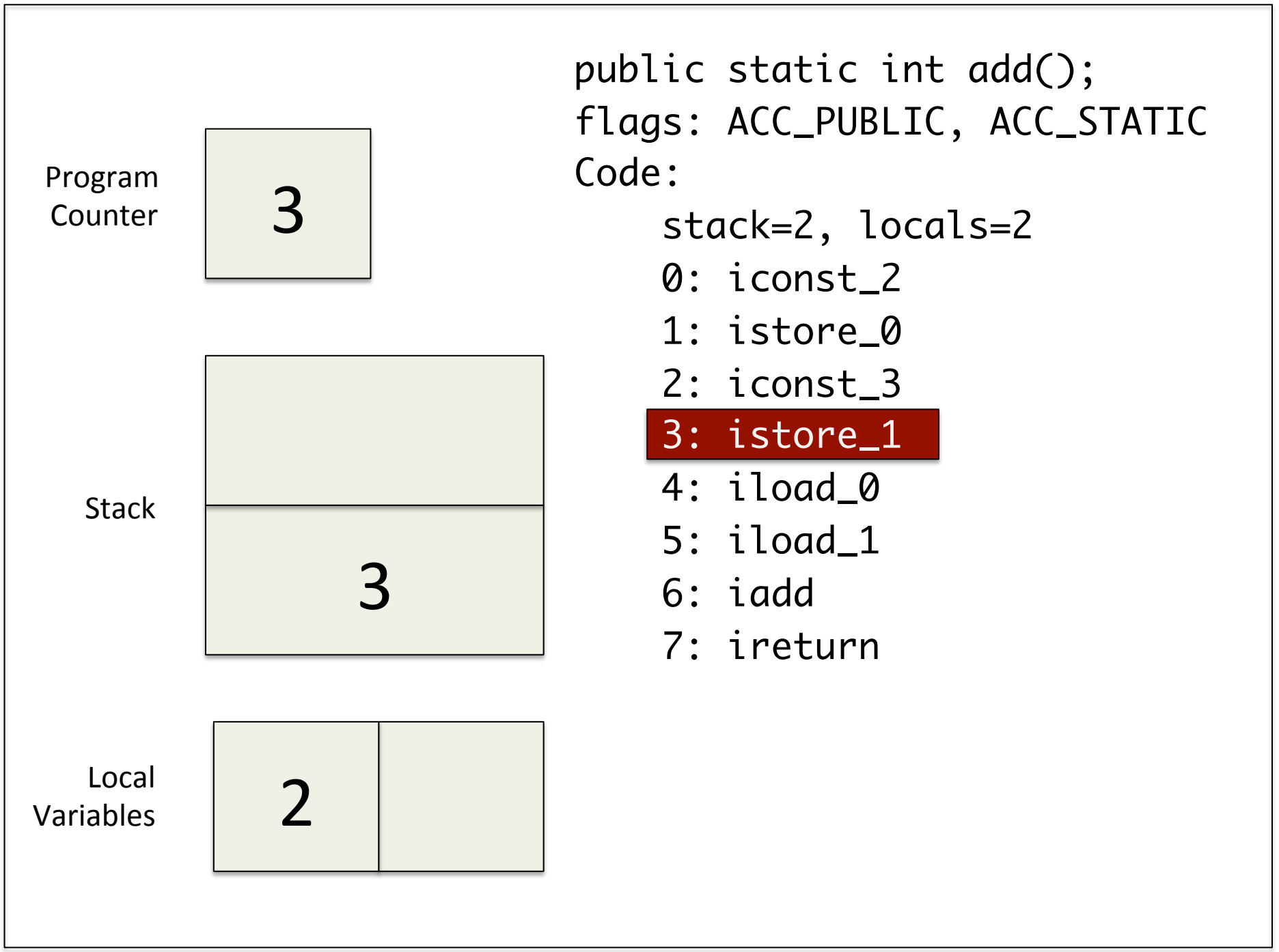
```java
public static int add() {
   final int i = 2;
   final int j = 3;
   return i + j;
 }
```

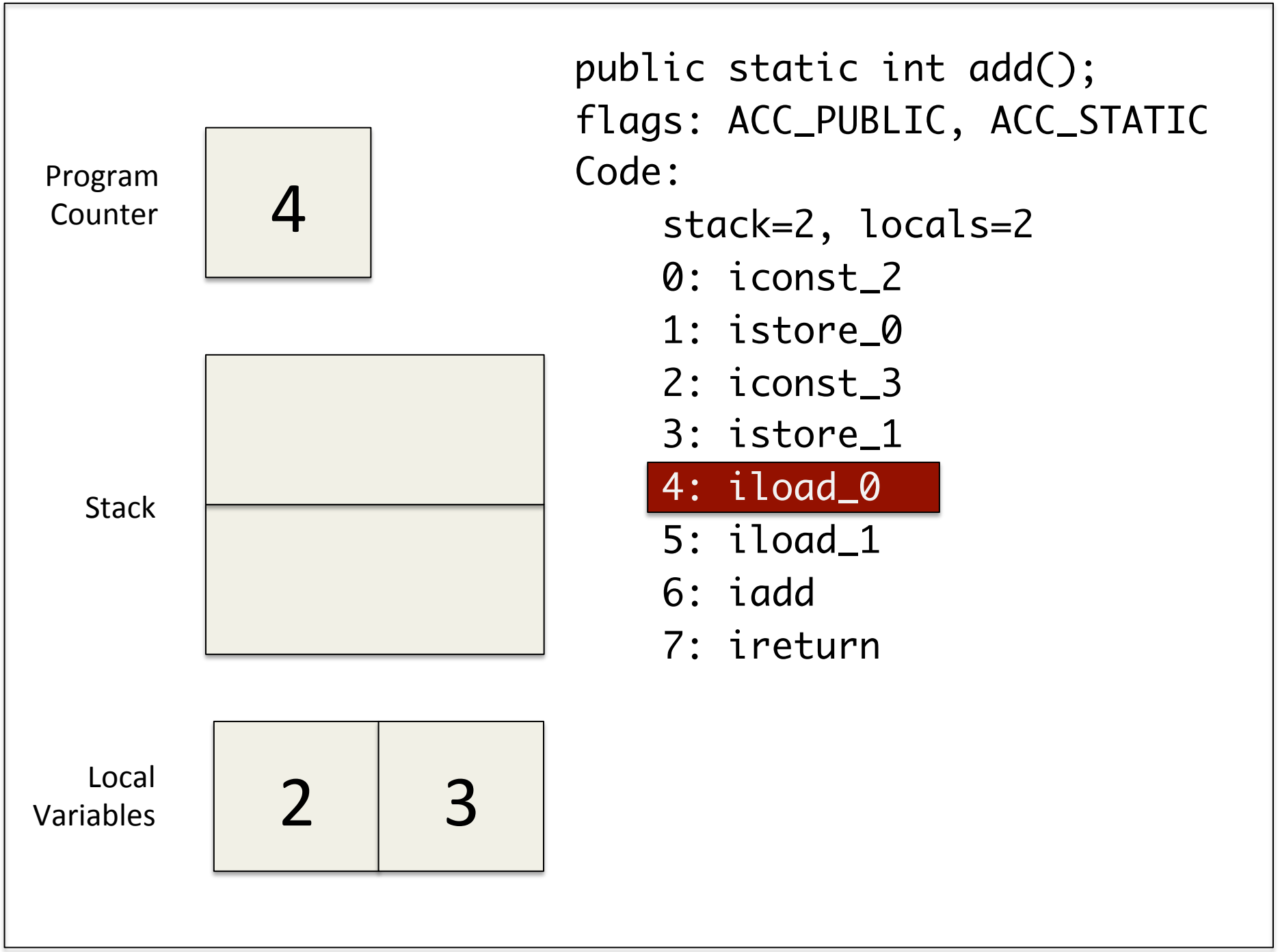
```
public static int add();
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=2
      0: iconst_2
      1: istore_0
      2: iconst_3
      3: istore_1
      4: iload_0
      5: iload_1
      6: iadd
      7: ireturn
```

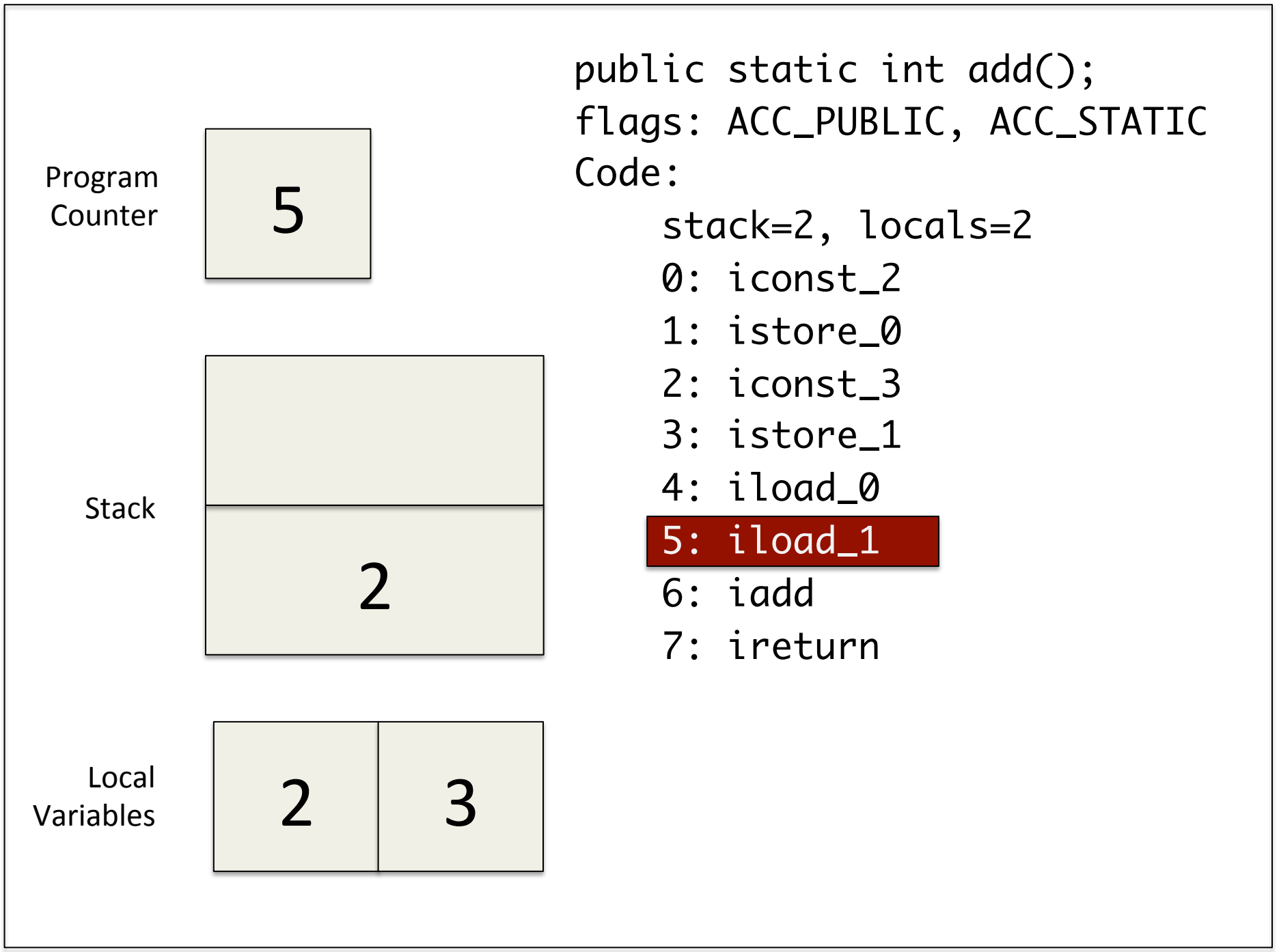

Program Counter
0
Stack
Local Variables
public static int add();
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=2
0: iconst_2
1: istore_0
2: iconst_3
3: istore_1
4: iload_0
5: iload_1
6: iadd
7: ireturn

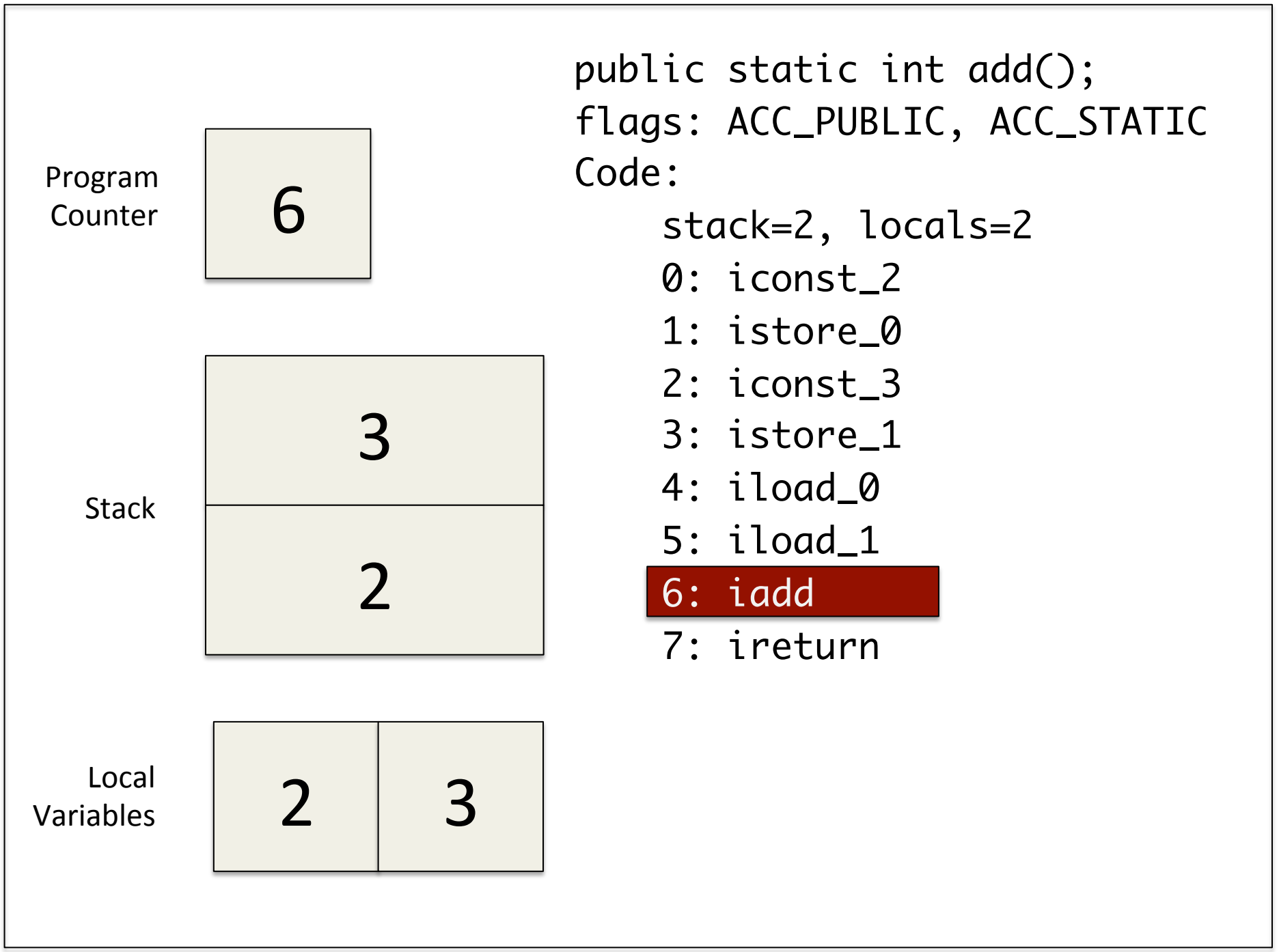

Program Counter
1
Stack
2
Local Variables
public static int add();
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=2
0: iconst_2
1: istore_0
2: iconst_3
3: istore_1
4: iload_0
5: iload_1
6: iadd
7: ireturn

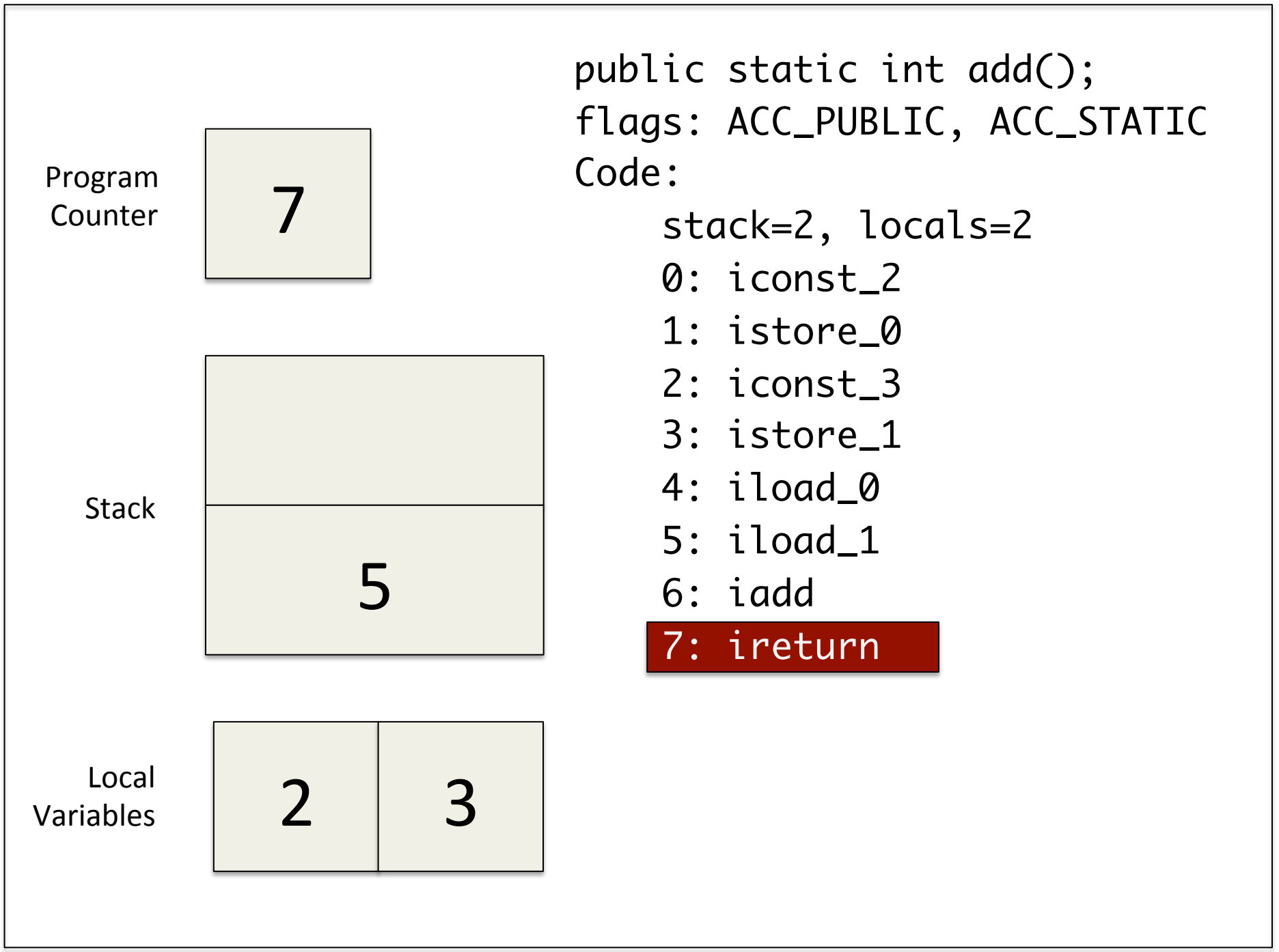

Program Counter
2
Stack
Local Variables
2
public static int add();
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=2
0: iconst_2
1: istore_0
2: iconst_3
3: istore_1
4: iload_0
5: iload_1
6: iadd
7: ireturn

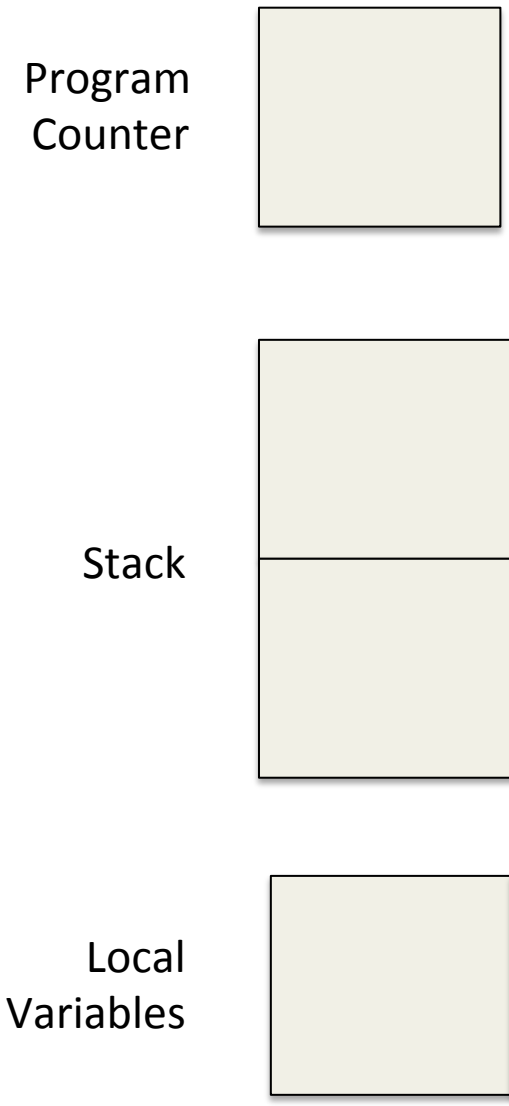

Program Counter
3
Stack
3
Local Variables
2
public static int add();
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=2
0: iconst_2
1: istore_0
2: iconst_3
3: istore_1
4: iload_0
5: iload_1
6: iadd
7: ireturn


Program Counter
4
Stack
Local Variables
2
3
public static int add();
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=2
0: iconst_2
1: istore_0
2: iconst_3
3: istore_1
4: iload_0
5: iload_1
6: iadd
7: ireturn

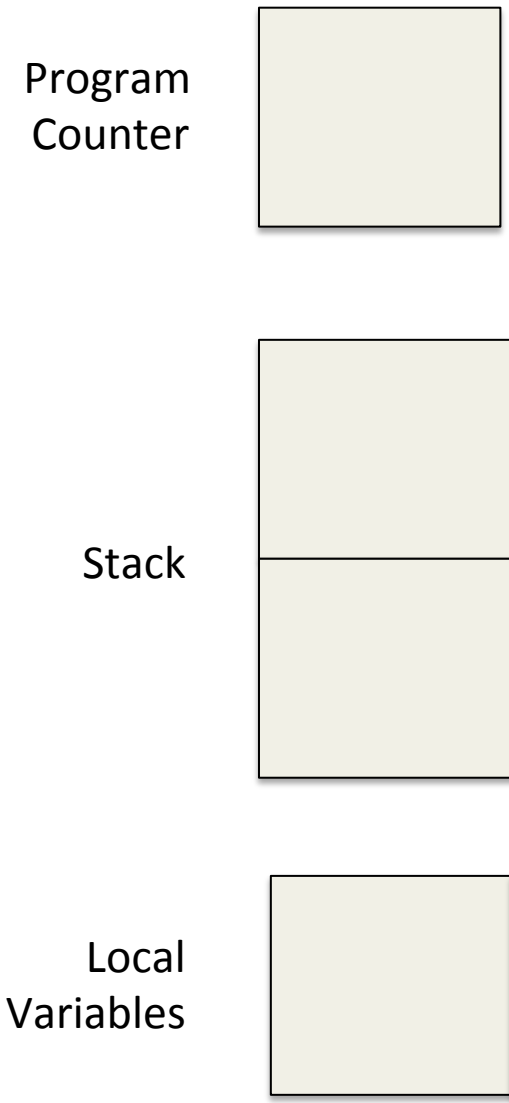

Program Counter
5
Stack
2
Local Variables
2
3
public static int add();
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=2
0: iconst_2
1: istore_0
2: iconst_3
3: istore_1
4: iload_0
5: iload_1
6: iadd
7: ireturn


Program Counter
6
Stack
3
2
Local Variables
2
3
public static int add();
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=2
0: iconst_2
1: istore_0
2: iconst_3
3: istore_1
4: iload_0
5: iload_1
6: iadd
7: ireturn

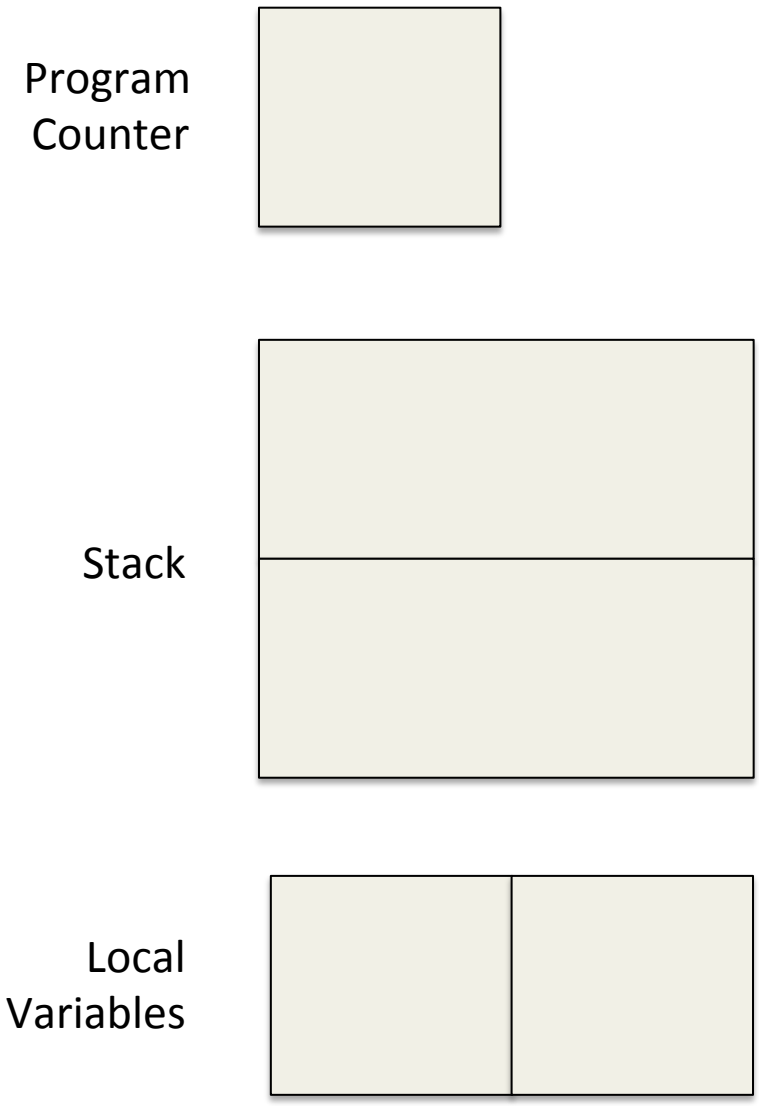

Program Counter
7
Stack
5
Local Variables
2
3
public static int add();
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=2
0: iconst_2
1: istore_0
2: iconst_3
3: istore_1
4: iload_0
5: iload_1
6: iadd
7: ireturn



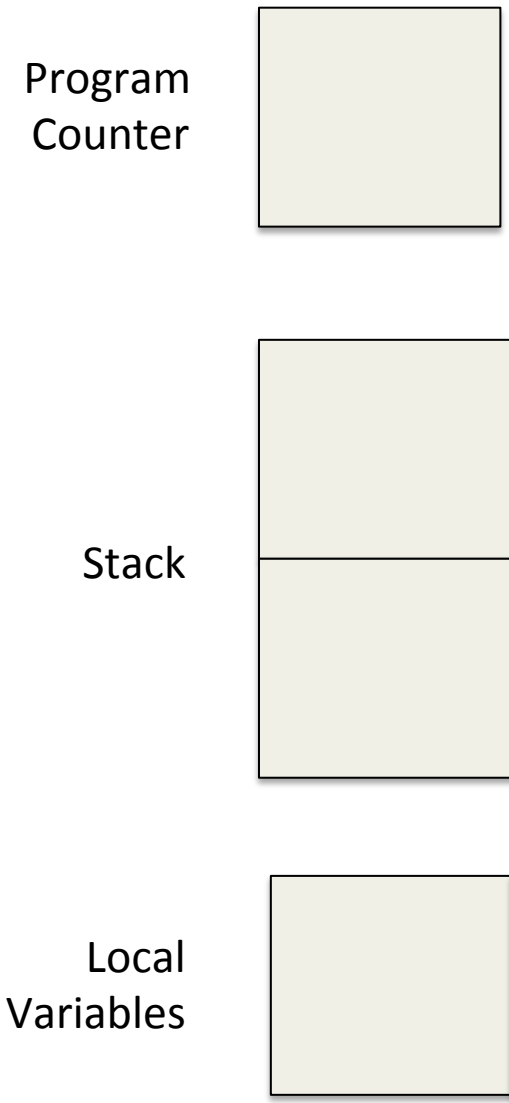

```
public static int add();
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=2, locals=2
    0: iconst_2
    1: istore_0
    2: iconst_3
    3: istore_1
    4: iload_0
    5: iload_1
    6: iadd
    7: ireturn
```



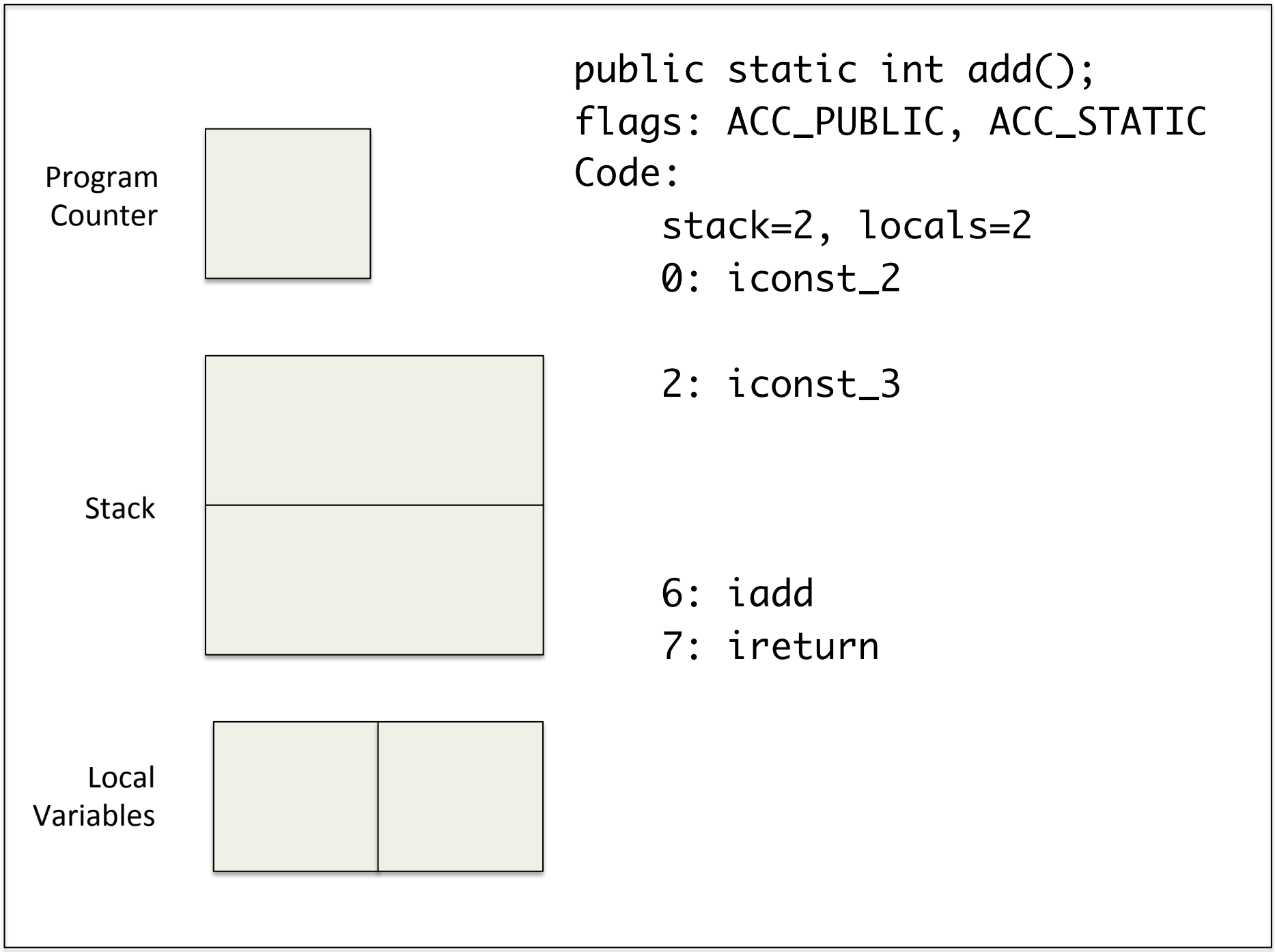

```
public static int add();
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=2, locals=2
    0: iconst_2
    1: istore_0
    2: iconst_3
    3: istore_1
    4: iload_0
    5: iload_1
    6: iadd
    7: ireturn
```

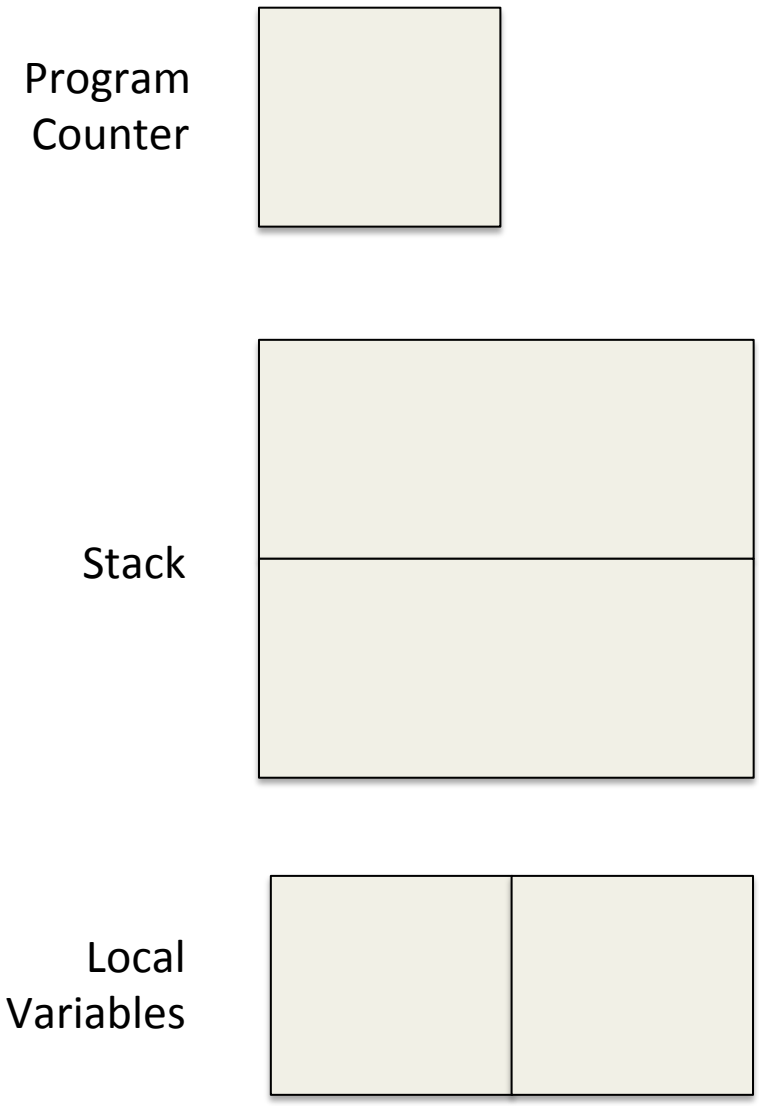



```
public static int add();
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=2, locals=2
    0: iconst_2
    1: istore_0
    2: iconst_3
    3: istore_1
    4: iload_0
    5: iload_1
    6: iadd
    7: ireturn
```

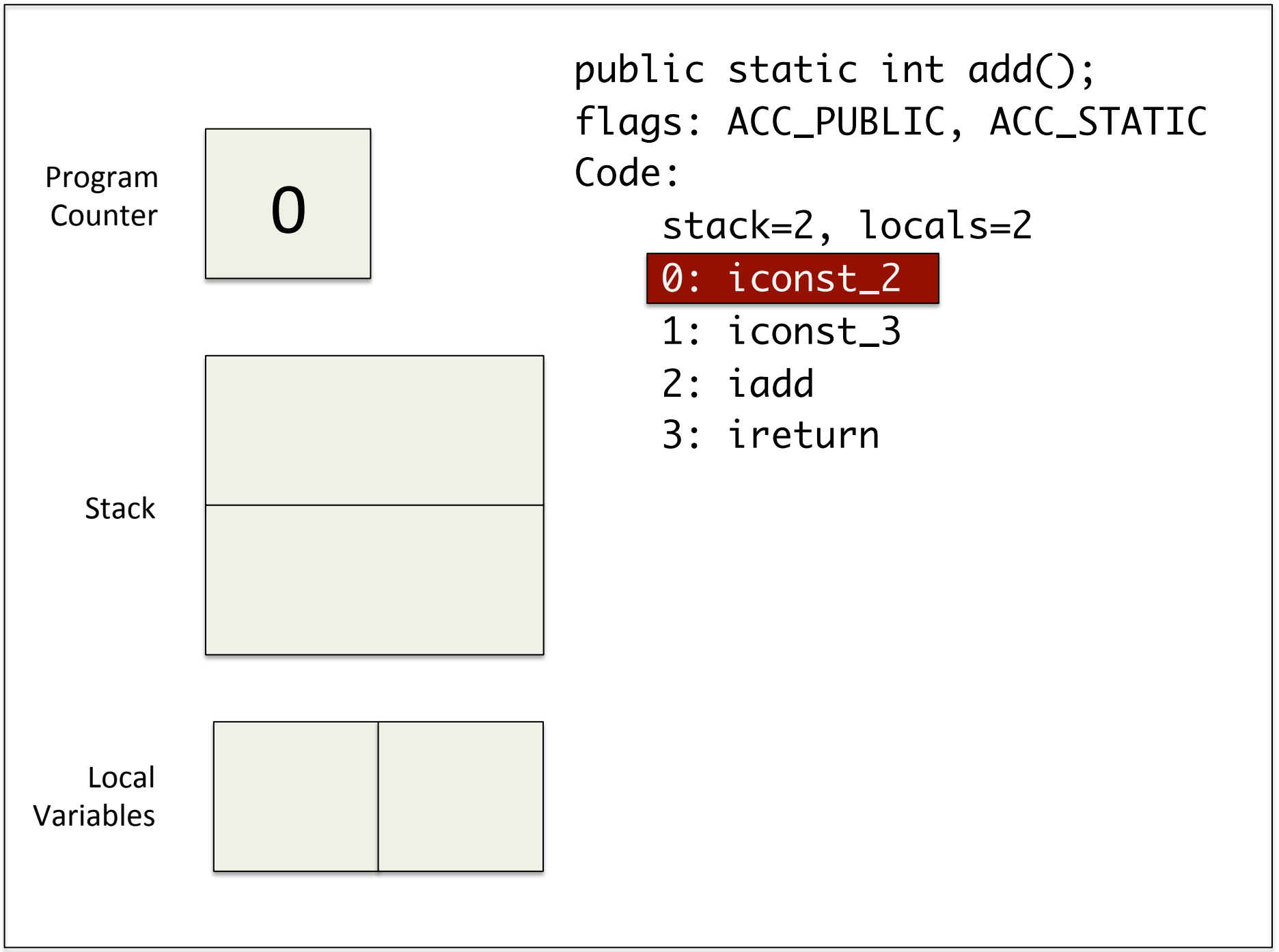


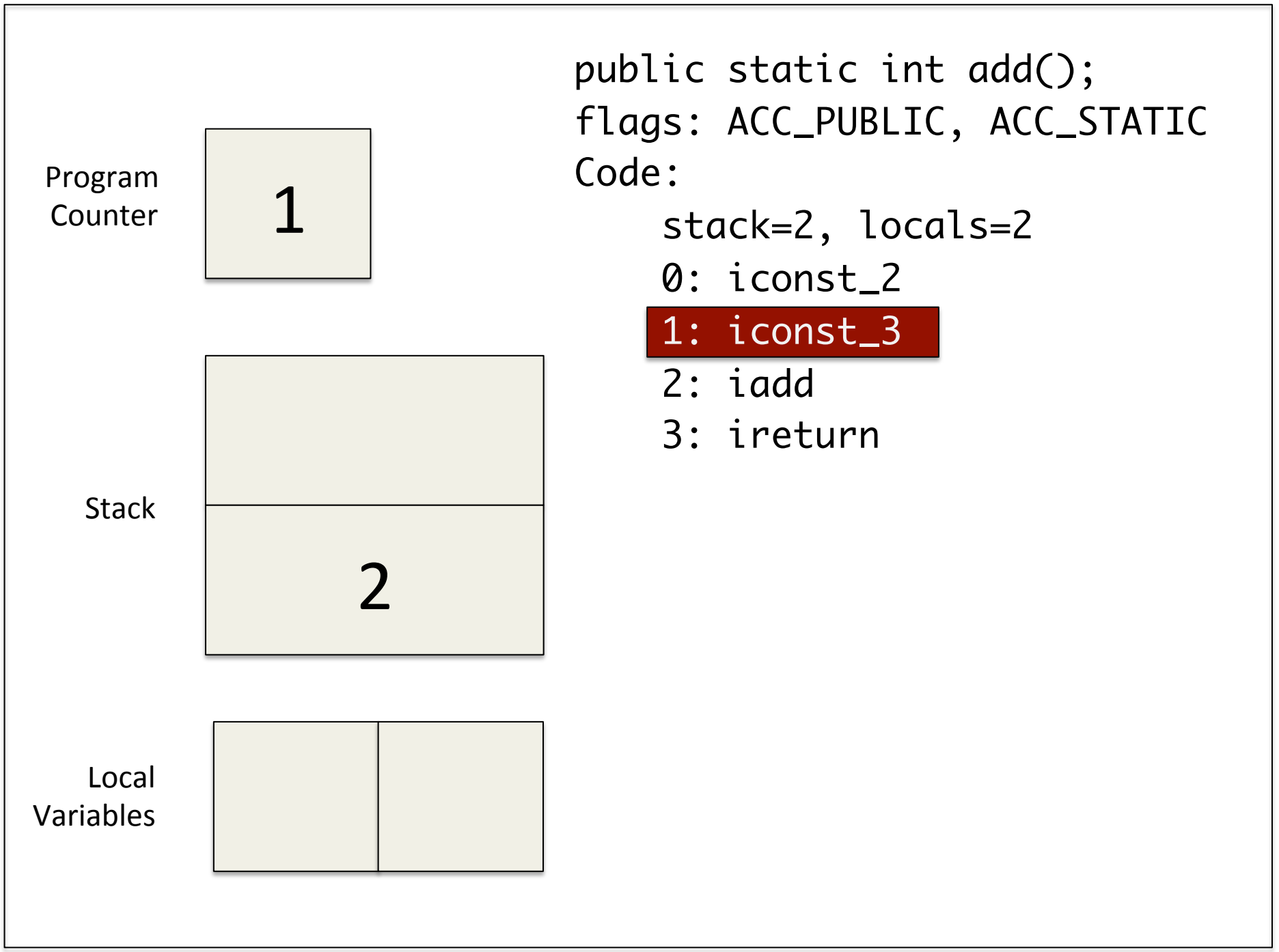

```
public static int add();
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=2, locals=2
    0: iconst_2
    1: istore_0
    2: iconst_3
    3: istore_1
    4: iload_0
    5: iload_1
    6: iadd
    7: ireturn
```

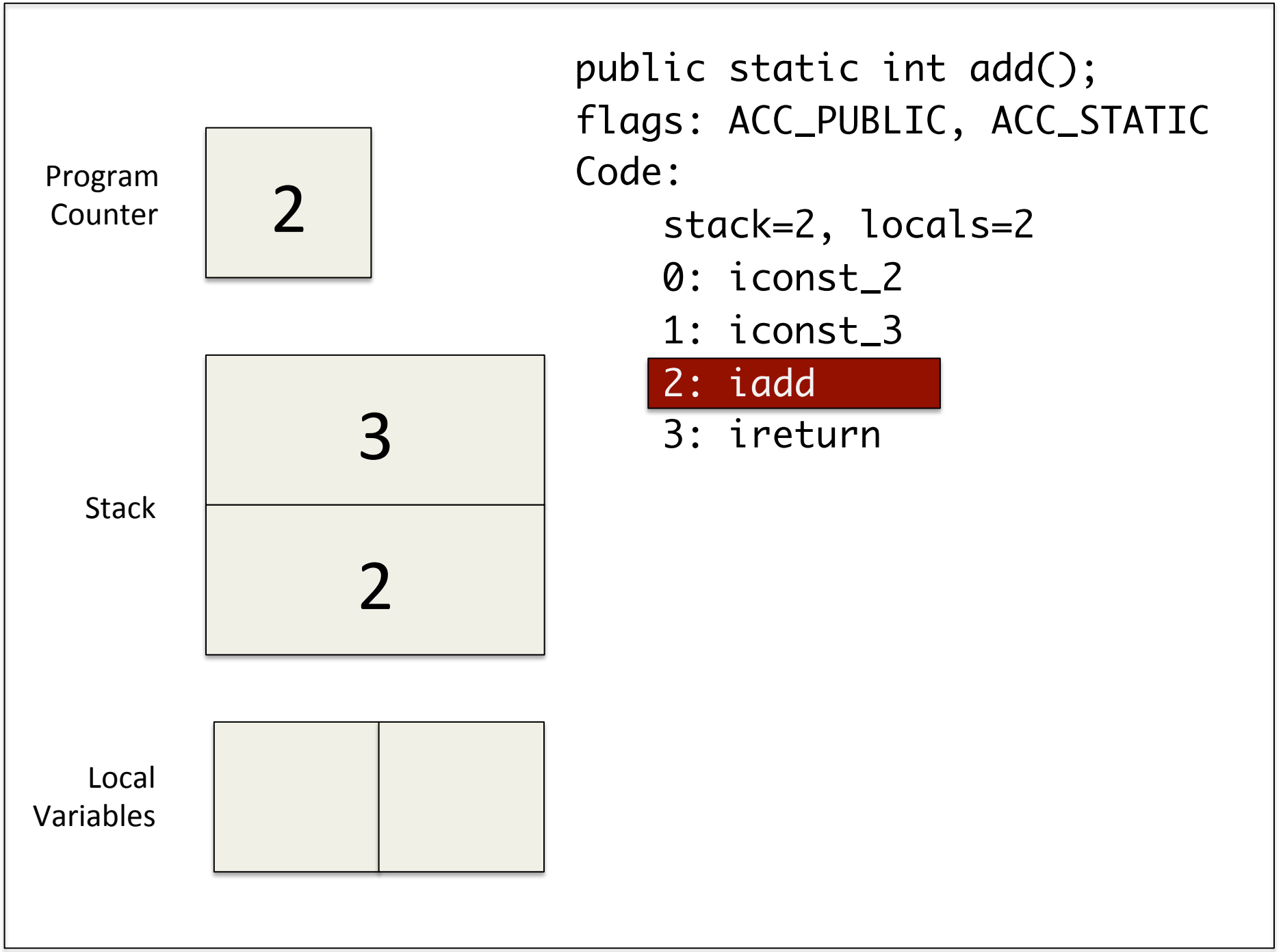

Program Counter
Stack
Local Variables
public static int add();
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=2
0: iconst_2
2: iconst_3
6: iadd
7: ireturn

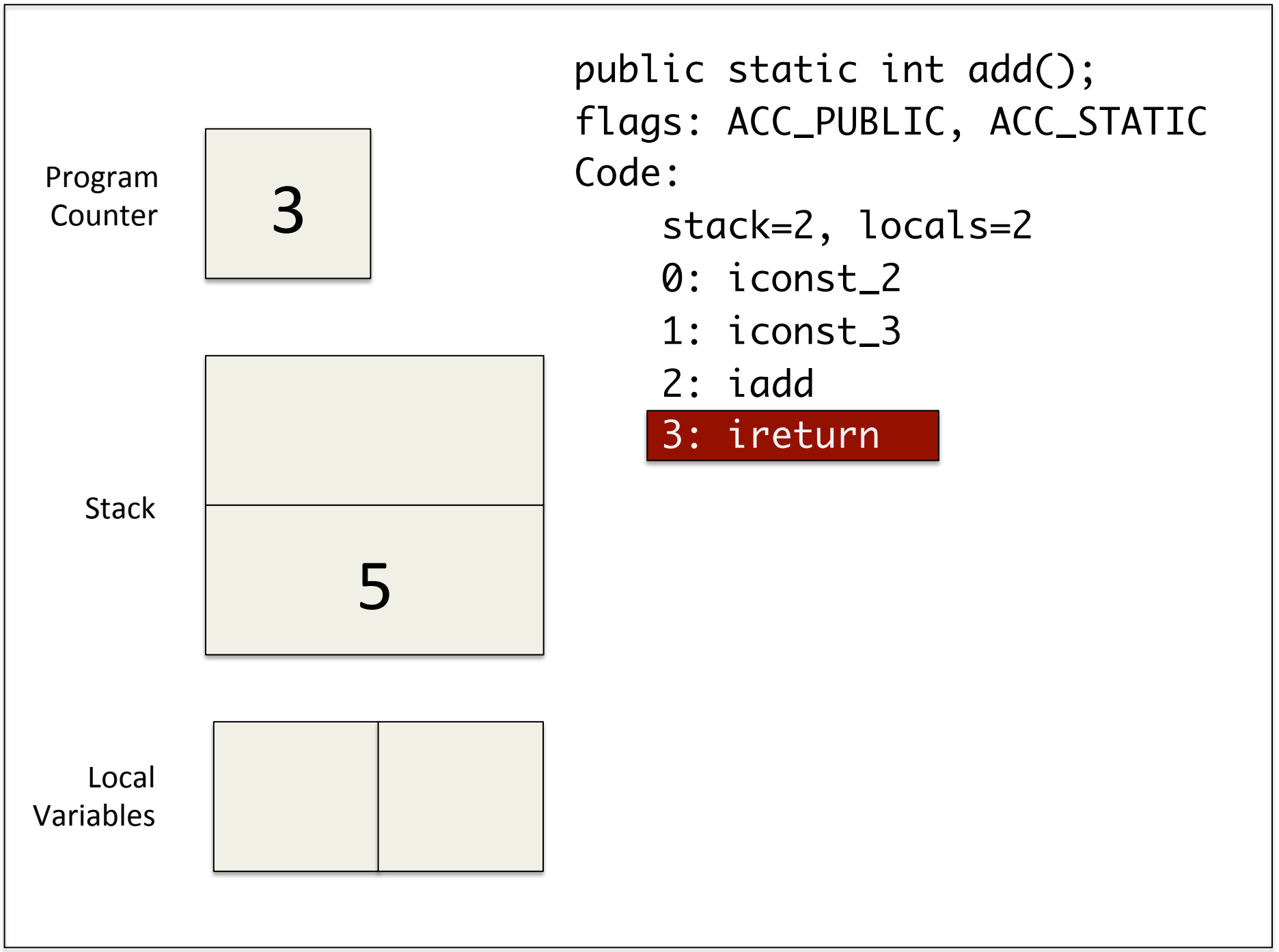


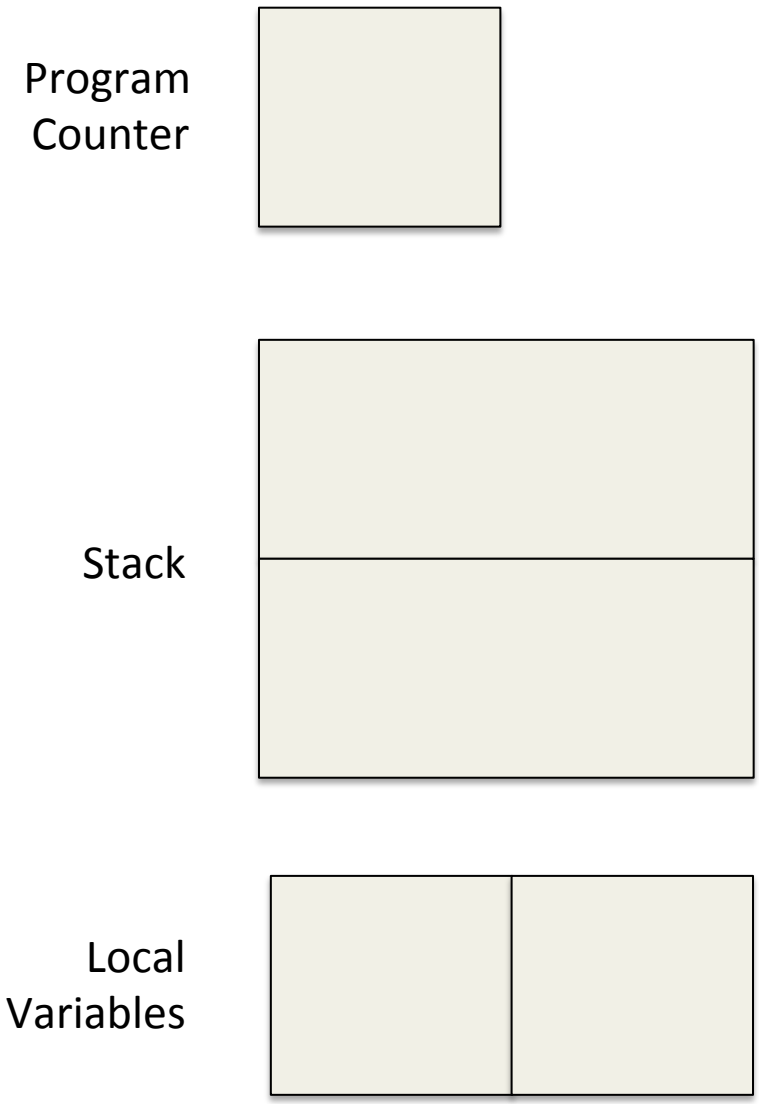

```
public static int add();
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=2, locals=2
    0: iconst_2
    1: iconst_3
    2: iadd
    3: ireturn
```


Program Counter
0
Stack
Local Variables
public static int add();
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=2
0: iconst_2
1: iconst_3
2: iadd
3: ireturn

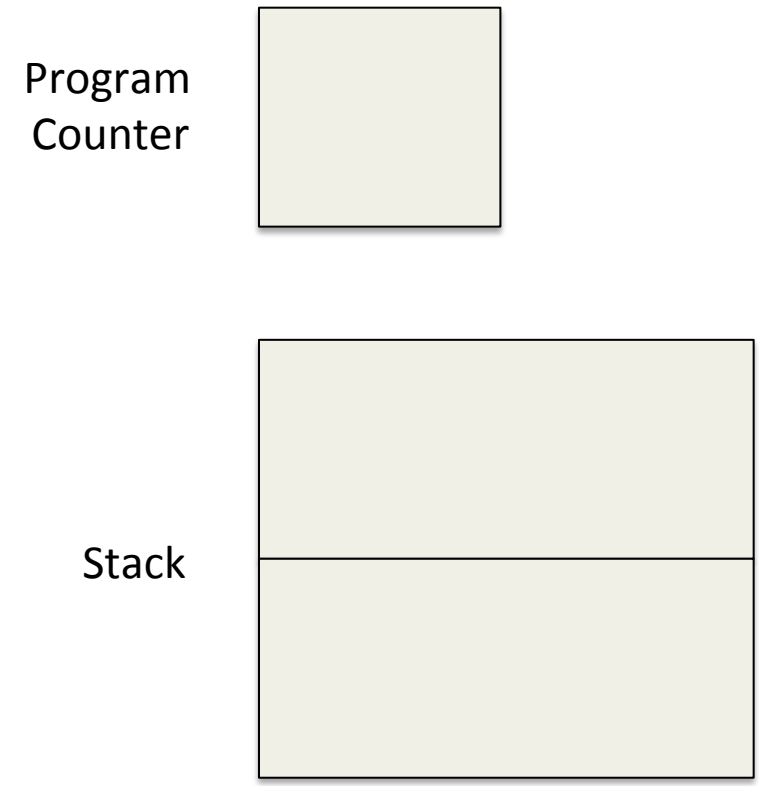

Program Counter
1
Stack
2
Local Variables
public static int add();
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=2
0: iconst_2
1: iconst_3
2: iadd
3: ireturn


Program Counter
2
Stack
3
2
Local Variables
public static int add();
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=2
0: iconst_2
1: iconst_3
2: iadd
3: ireturn


Program Counter
3
Stack
5
Local Variables
public static int add();
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=2
0: iconst_2
1: iconst_3
2: iadd
3: ireturn




```
public static int add();
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=2, locals=2
    0: iconst_2
    1: iconst_3
    2: iadd
    3: ireturn
```




```
public static int add();
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=2, locals=0
    0: iconst_2
    1: iconst_3
    2: iadd
    3: ireturn
```

Loop Unrolling

- Compare and jump pattern can be expensive
  - Involves loads and stores
  - Blows out the I-Cache
- Transform loops with known iterations
  - Turn into linear code without jumps
- Exploit program knowledge
  - Constant folding
  - Value analysis

```
public int loop() {
   int result = 0;
   for (int i = 0; i < 5; i++) {
     result += i;
   }
   return result;
}
```

```
public int loop() {
   int result = 0;
   for (int i = 0; i < 5; i++) {
     result += i;
   }
   return result;
}
```

```
public int loop() {
   int result = 0;
   result += 0;
   result += 1;
   result += 2;
   result += 3;
   result += 4;
   return result;
}
```

Loop Unrolling Performance

- How many times do you unroll?
- More unrolling means fewer jumps
- More instructions hurt the I-Cache
- Trade-off based on the target platform

Register Allocation

- Decide what values to store where
  - Loop indices should be in registers
- Spill registers if not enough
- Linear Scan register allocation
  - Quick and easy
  - May lead to more spilling
- Graph Coloring
  - More complex
  - Can generate a more optimal pattern

Type Inference

- Objects can be referred to by super type
  - The dynamic type is known only at allocation
- We can sometimes figure out a variable's type
  - Depends on the code paths to the site
- May be optimistic
  - If so, we need a fallback to maintain correctness

```
class ClassA {

    public void method() {
        final Object a = new ClassA();
        final Object obj = new ClassB((ClassA) a);
        printClass(obj);
    }

    private void printClass(final Object obj) {
        System.out.println(obj.getClass());
    }
}

class ClassB {
    public final ClassA a;

    public ClassB(final ClassA a) {
        this.a = a;
    }
}
```

```
class ClassA {

    public void method() {
        final Object a = new ClassA();
        final Object obj = new ClassB((ClassA) a);
        printClass(obj);
    }

    private void printClass(final Object obj) {
        System.out.println(obj.getClass());
    }
}

class ClassB {
    public final ClassA a;

    public ClassB(final ClassA a) {
        this.a = a;
    }
}
```

```
class ClassA {

    public void method() {
        final ClassA a = new ClassA();
        final Object obj = new ClassB((ClassA) a);
        printClass(obj);
    }

    private void printClass(final Object obj) {
        System.out.println(obj.getClass());
    }
}

class ClassB {
    public final ClassA a;

    public ClassB(final ClassA a) {
        this.a = a;
    }
}
```

```
class ClassA {

    public void method() {
        final ClassA a = new ClassA();
        final Object obj = new ClassB((ClassA) a);
        printClass(obj);
    }

    private void printClass(final Object obj) {
        System.out.println(obj.getClass());
    }
}

class ClassB {
    public final ClassA a;

    public ClassB(final ClassA a) {
        this.a = a;
    }
}
```

```
class ClassA {

    public void method() {
        final ClassA a = new ClassA();
        final Object obj = new ClassB(a);
        printClass(obj);
    }

    private void printClass(final Object obj) {
        System.out.println(obj.getClass());
    }
}

class ClassB {
    public final ClassA a;

    public ClassB(final ClassA a) {
        this.a = a;
    }
}
```

```
class ClassA {

    public void method() {
        final ClassA a = new ClassA();
        final Object obj = new ClassB(a);
        printClass(obj);
    }

    private void printClass(final Object obj) {
        System.out.println(obj.getClass());
    }
}

class ClassB {
    public final ClassA a;

    public ClassB(final ClassA a) {
        this.a = a;
    }
}
```

```
class ClassA {

    public void method() {
        final ClassA a = new ClassA();
        final ClassB obj = new ClassB(a);
        printClass(obj);
    }

    private void printClass(final Object obj) {
        System.out.println(obj.getClass());
    }
}

class ClassB {
    public final ClassA a;

    public ClassB(final ClassA a) {
        this.a = a;
    }
}
```

```
class ClassA {

    public void method() {
        final ClassA a = new ClassA();
        final ClassB obj = new ClassB(a);
        printClass(obj);
    }

    private void printClass(final Object obj) {
        System.out.println(obj.getClass());
    }
}

class ClassB {
    public final ClassA a;

    public ClassB(final ClassA a) {
        this.a = a;
    }
}
```

```
class ClassA {

    public void method() {
        final ClassA a = new ClassA();
        final ClassB obj = new ClassB(a);
        printClass(obj);
    }

    private void printClass(final ClassB obj) {
        System.out.println(obj.getClass());
    }
}

class ClassB {
    public final ClassA a;

    public ClassB(final ClassA a) {
        this.a = a;
    }
}
```

```
class ClassA {

    public void method() {
        final ClassA a = new ClassA();
        final ClassB obj = new ClassB(a);
        printClass(obj);
    }

    private void printClass(final ClassB obj) {
        System.out.println(obj.getClass());
    }
}

class ClassB {
    public final ClassA a;

    public ClassB(final ClassA a) {
        this.a = a;
    }
}
```

```
class ClassA {

    public void method() {
        final ClassA a = new ClassA();
        final ClassB obj = new ClassB(a);
        printClass(obj);
    }

    private void printClass(final ClassB obj) {
        System.out.println(obj.getClass());
    }
}

class ClassB {
    public final ClassA a;

    public ClassB(final ClassA a) {
        this.a = a;
    }
}
```

```
class ClassA {

    public void method() {
        final ClassA a = new ClassA();
        final ClassB obj = new ClassB(a);
        printClass(obj);
    }

    public  void printClass(final ClassB obj) {
        System.out.println(obj.getClass());
    }
}

class ClassB {
    public final ClassA a;

    public ClassB(final ClassA a) {
        this.a = a;
    }
}
```

Devirtualization

- Virtual dispatch of methods is expensive
  - Need to traverse the class hierarchy
- Static calls are much faster

- Replace `invokedynamic` with `invokestatic`
  - If there's only one possible method to call
  - We can tell from the class hierarchy analysis

- May have to be undone if the CHA changes

```
class ClassA {

    public void method() {
        final Object a = new ClassA();
        final Object obj = new ClassB((ClassA) a);
        printClass();
    }

    void printClass() {
        System.out.println(this.getClass());
    }
}
```

```
class ClassA {

    public void method() {
        final Object a = new ClassA();
        final Object obj = new ClassB((ClassA) a);
        printClass();
    }

    void printClass() {
        System.out.println(this.getClass());
    }
}
```

```
class ClassA {

    public void method() {
        final Object a = new ClassA();
        final Object obj = new ClassB((ClassA) a);
        this.printClass();
    }

    void printClass() {
        System.out.println(this.getClass());
    }
}
```

```
class ClassA {

    public void method() {
        final Object a = new ClassA();
        final Object obj = new ClassB((ClassA) a);
        this.printClass();
    }

    void printClass() {
        System.out.println(this.getClass());
    }
}
```

```
class ClassA {

    public void method() {
        final Object a = new ClassA();
        final Object obj = new ClassB((ClassA) a);
        this.printClass();
    }

    static void printClass() {
        System.out.println(this.getClass());
    }
}
```

```
class ClassA {

    public void method() {
        final Object a = new ClassA();
        final Object obj = new ClassB((ClassA) a);
        this.printClass();
    }

    static void printClass() {
        System.out.println(this.getClass());
    }
}
```

```
class ClassA {

    public void method() {
        final Object a = new ClassA();
        final Object obj = new ClassB((ClassA) a);
        ClassA.printClass();
    }

    static void printClass() {
        System.out.println(this.getClass());
    }
}
```

```
class ClassA {

    public void method() {
        final Object a = new ClassA();
        final Object obj = new ClassB((ClassA) a);
        ClassA.printClass();
    }

    static void printClass() {
        System.out.println(this.getClass());
    }
}
```

```
class ClassA {

    public void method() {
        final Object a = new ClassA();
        final Object obj = new ClassB((ClassA) a);
        ClassA.printClass();
    }

    static void printClass(final ClassA thisPtr) {
        System.out.println(thisPtr.getClass());
    }
}
```

```
class ClassA {

    public void method() {
        final Object a = new ClassA();
        final Object obj = new ClassB((ClassA) a);
        ClassA.printClass();
    }

    static void printClass(final ClassA thisPtr) {
        System.out.println(thisPtr.getClass());
    }
}
```

```
class ClassA {

    public void method() {
        final Object a = new ClassA();
        final Object obj = new ClassB((ClassA) a);
        ClassA.printClass(this);
    }

    static void printClass(final ClassA thisPtr) {
        System.out.println(thisPtr.getClass());
    }
}
```

```
class ClassA {

    public void method() {
        final Object a = new ClassA();
        final Object obj = new ClassB((ClassA) a);
        ClassA.printClass(this);
    }

    static void printClass(final ClassA thisPtr) {
        System.out.println(thisPtr.getClass());
    }
}
```

```
class ClassA {

    public void method() {
        final Object a = new ClassA();
        final Object obj = new ClassB((ClassA) a);
        ClassA.printClass(this);
    }

    static void printClass(final ClassA thisPtr) {
        System.out.println(thisPtr.getClass());
    }
}
```

```
class ClassA {

    public void method() {
        final Object a = new ClassA();
        final Object obj = new ClassB((ClassA) a);
        ClassA.printClass(this);
    }

    static void printClass(final ClassA thisPtr) {
        System.out.println(thisPtr.getClass());
    }
}



class ClassC {
    void printClass() {
        System.out.println("Take that, optimization");
    }
}
```

```
class ClassA {

    public void method() {
        final Object a = new ClassA();
        final Object obj = new ClassB((ClassA) a);
        printClass();
    }

    void printClass() {
        System.out.println(this.getClass());
    }
}



class ClassC {
    void printClass() {
        System.out.println("Take that, optimization");
    }
}
```

Inlining

- Any method invocation is expensive
  - Finding the correct method to call
  - Transfer control to new stack frame
  - Need to save register and PC values
  - I-Cache effects
- Inlining inserts method body at call site
  - Call site must be monomorphic

```
static int multiply(final int x, final int y) {
    int product = 0;
    for (int i = 0; i < y; i++) {
        product = add(product, x);
    }
    return product;
 }


 static int add(final int a, final int b) {
     return a + b;
 }
```

```
static int multiply(final int x, final int y) {
    int product = 0;
    for (int i = 0; i < y; i++) {
        product = add(product, x);
    }
    return product;
}

static int add(final int a, final int b) {
    return a + b;
}
```

```
static int multiply(final int x, final int y) {
    int product = 0;
    for (int i = 0; i < y; i++) {
        product = add(product, x);
    }
    return product;
}

static int add(final int a, final int b) {
    return a + b;
}
```

```
static int multiply(final int x, final int y) {
    int product = 0;
    for (int i = 0; i < y; i++) {
        product = a + b;
    }
    return product;
 }


 static int add(final int a, final int b) {
     return a + b;
 }
```

```
static int multiply(final int x, final int y) {
    int product = 0;
    for (int i = 0; i < y; i++) {
        product = a + b;
    }
    return product;
 }


 static int add(final int a, final int b) {
     return a + b;
 }
```

```
static int multiply(final int x, final int y) {
    int product = 0;
    for (int i = 0; i < y; i++) {
        final int a = product;
        final int b = x;
        product = a + b;
    }
    return product;
}

static int add(final int a, final int b) {
    return a + b;
}
```

```
static int multiply(final int x, final int y) {
    int product = 0;
    for (int i = 0; i < y; i++) {
        final int a = product;
        final int b = x;
        product = a + b;
    }
    return product;
}

static int add(final int a, final int b) {
    return a + b;
}
```

```
static int multiply(final int x, final int y) {
    int product = 0;
    for (int i = 0; i < y; i++) {
        product = product + x;
    }
    return product;
 }


 static int add(final int a, final int b) {
     return a + b;
 }
```

Inlining

- Inlining normally the most valuable optimization
  - Particularly when combined with devirtualization
- Gives larger block of code to optimize
  - Other optimizations become more useful
  - Think about reducing redundant loads/stores
- We can inline repeatedly
  - Methods become larger
  - Better I-Cache utilization

When to Inline

- Decisions based partly on CHA
  - Helps to identify monomorphic call sites
- Biased by bytecode size
  - Inlining small methods saves on call overhead
- Helps to build up larger methods
  - More room for optimization
- Inlining may be optimistic
  - Include a guard in case the callsite becomes polymorphic

```
public class MainClass {

    public static final int N = 7;

    public static void main(final String[] args) {
        int answer = 0;
        for (int i = 0; i < N - 1; i++) {
          answer = add(answer, N);
        }
        for (int i = 0; i < 1000000; i++) {
            if (i % 2 == 0) {
                answer = add(answer, N);
            } else {
                answer = add(answer, (N * -1));
            }
        }
        System.out.println("The answer is " + answer);
    }
}
```

```
public class MainClass {

    public static final int N = 7;

    public static void main(final String[] args) {
        int answer = 0;
        for (int i = 0; i < N - 1; i++) {
          answer = add(answer, N);
        }
        for (int i = 0; i < 1000000; i++) {
            if (i % 2 == 0) {
                answer = add(answer, N);
            } else {
                answer = add(answer, (N * -1));
            }
        }
        System.out.println("The answer is " + answer);
    }
}
```

```
public class MainClass {

    public static final int N = 7;

    public static void main(final String[] args) {
        int answer = 0;
        for (int i = 0; i < 7 - 1; i++) {
          answer = add(answer, 7);
        }
        for (int i = 0; i < 1000000; i++) {
            if (i % 2 == 0) {
                answer = add(answer, 7);
            } else {
                answer = add(answer, (7 * -1));
            }
        }
        System.out.println("The answer is " + answer);
    }
}
```

```
public class MainClass {

    public static final int N = 7;

    public static void main(final String[] args) {
        int answer = 0;
        for (int i = 0; i < 7 - 1; i++) {
          answer = add(answer, 7);
        }
        for (int i = 0; i < 1000000; i++) {
            if (i % 2 == 0) {
                answer = add(answer, 7);
            } else {
                answer = add(answer, (7 * -1));
            }
        }
        System.out.println("The answer is " + answer);
    }
}
```

```
public class MainClass {

    public static final int N = 7;

    public static void main(final String[] args) {
        int answer = 0;
        for (int i = 0; i < 6; i++) {
          answer = add(answer, 7);
        }
        for (int i = 0; i < 1000000; i++) {
            if (i % 2 == 0) {
                answer = add(answer, 7);
            } else {
                answer = add(answer, (-7));
            }
        }
        System.out.println("The answer is " + answer);
    }
}
```

```
public class MainClass {

    public static final int N = 7;

    public static void main(final String[] args) {
        int answer = 0;
        for (int i = 0; i < 6; i++) {
          answer = add(answer, 7);
        }
        for (int i = 0; i < 1000000; i++) {
            if (i % 2 == 0) {
                answer = add(answer, 7);
            } else {
                answer = add(answer, (-7));
            }
        }
        System.out.println("The answer is " + answer);
    }
}
```

```
public class MainClass {

    public static final int N = 7;

    public static void main(final String[] args) {
        int answer = 0;
        answer = add(answer, 7);
        answer = add(answer, 7);
        answer = add(answer, 7);
        answer = add(answer, 7);
        answer = add(answer, 7);
        answer = add(answer, 7);
        for (int i = 0; i < 1000000; i++) {
            if (i % 2 == 0) {
                answer = add(answer, 7);
            } else {
                answer = add(answer, (-7));
            }
        }
        System.out.println("The answer is " + answer);
    }
}
```

```
public class MainClass {

    public static final int N = 7;

    public static void main(final String[] args) {
        int answer = 0;
        answer = add(answer, 7);
        answer = add(answer, 7);
        answer = add(answer, 7);
        answer = add(answer, 7);
        answer = add(answer, 7);
        answer = add(answer, 7);
        for (int i = 0; i < 1000000; i++) {
            if (i % 2 == 0) {
                answer = add(answer, 7);
            } else {
                answer = add(answer, (-7));
            }
        }
        System.out.println("The answer is " + answer);
    }
}
```

```
public class MainClass {

    public static final int N = 7;

    public static void main(final String[] args) {
        int answer = 0;
        answer = add(answer, 7);
        answer = add(answer, 7);
        answer = add(answer, 7);
        answer = add(answer, 7);
        answer = add(answer, 7);
        answer = add(answer, 7);
        for (int i = 0; i < 50000; i++) {
            if (i % 2 == 0) { answer = add(answer, 7);
            } else          { answer = add(answer, (-7));}
            if (i % 2 == 0) { answer = add(answer, 7);
            } else          { answer = add(answer, (-7));}
        }
        System.out.println("The answer is " + answer);
    }
}
```

```
public class MainClass {

    public static final int N = 7;

    public static void main(final String[] args) {
        int answer = 0;
        answer = add(answer, 7);
        answer = add(answer, 7);
        answer = add(answer, 7);
        answer = add(answer, 7);
        answer = add(answer, 7);
        answer = add(answer, 7);
        for (int i = 0; i < 50000; i++) {
            answer = add(answer, 7);
            answer = add(answer, (-7));
        }
        System.out.println("The answer is " + answer);
    }
}
```

```
public class MainClass {

    public static final int N = 7;

    public static void main(final String[] args) {
        int answer = 0;
        answer = add(answer, 7);
        answer = add(answer, 7);
        answer = add(answer, 7);
        answer = add(answer, 7);
        answer = add(answer, 7);
        answer = add(answer, 7);
        for (int i = 0; i < 50000; i++) {
            answer = add(answer, 7);
            answer = add(answer, (-7));
        }
        System.out.println("The answer is " + answer);
    }
}
```

```
public class MainClass {

    public static final int N = 7;

    public static void main(final String[] args) {
        int answer = 0;
        answer = answer + 7;
        answer = answer + 7;
        answer = answer + 7;
        answer = answer + 7;
        answer = answer + 7;
        answer = answer + 7;
        for (int i = 0; i < 50000; i++) {
            answer = answer + 7;
            answer = answer + (-7);
        }
        System.out.println("The answer is " + answer);
    }
}
```

```
public class MainClass {

    public static final int N = 7;

    public static void main(final String[] args) {
        int answer = 0;
        answer = answer + 7;
        answer = answer + 7;
        answer = answer + 7;
        answer = answer + 7;
        answer = answer + 7;
        answer = answer + 7;
        for (int i = 0; i < 50000; i++) {
            answer = answer + 7;
            answer = answer + (-7);
        }
        System.out.println("The answer is " + answer);
    }
}
```

```
public class MainClass {

    public static final int N = 7;

    public static void main(final String[] args) {
        int answer = 0 + 7 + 7 + 7 + 7 + 7 + 7
        for (int i = 0; i < 50000; i++) {
            answer = answer + 7;
            answer = answer + (-7);
        }
        System.out.println("The answer is " + answer);
    }
}
```

```
public class MainClass {

    public static final int N = 7;

    public static void main(final String[] args) {
        int answer = 42
        for (int i = 0; i < 50000; i++) {
            answer = answer + 7;
            answer = answer + (-7);
        }
        System.out.println("The answer is " + answer);
    }
}
```

```
public class MainClass {

    public static final int N = 7;

    public static void main(final String[] args) {
        int answer = 42
        for (int i = 0; i < 50000; i++) {
            answer = answer + 7;
            answer = answer + (-7);
        }
        System.out.println("The answer is " + answer);
    }
}
```

```
public class MainClass {

    public static final int N = 7;

    public static void main(final String[] args) {
        int answer = 42
        for (int i = 0; i < 50000; i++) {
            answer = answer + 7 - 7;
        }
        System.out.println("The answer is " + answer);
    }
}
```

```
public class MainClass {

    public static final int N = 7;

    public static void main(final String[] args) {
        int answer = 42
        for (int i = 0; i < 50000; i++) {
            answer = answer + 0;
        }
        System.out.println("The answer is " + answer);
    }
}
```

```
public class MainClass {

    public static final int N = 7;

    public static void main(final String[] args) {
        int answer = 42
        for (int i = 0; i < 50000; i++) {
            answer = answer + 0;
        }
        System.out.println("The answer is " + answer);
    }
}
```

```
public class MainClass {

    public static final int N = 7;

    public static void main(final String[] args) {
        int answer = 42
        System.out.println("The answer is " + answer);
    }
}
```

```
public class MainClass {

    public static final int N = 7;

    public static void main(final String[] args) {
        int answer = 42
        System.out.println("The answer is " + answer);
    }
}
```

```
public class MainClass {

    public static final int N = 7;

    public static void main(final String[] args) {
        System.out.println("The answer is " + 42);
    }
}
```

On-Stack Replacement

- Sometimes a hot method never exits
  - No chance to compile for next invocation
- Compile code and replace
- Tricky to implement
- Questionable real-word applicability
  - Single hot method not a common pattern

Intrinsics

- Behavior of some methods are well known
  - `System.arraycopy`
  - `Object.hashCode`
- Compiler ignores the class file
  - Provides its own optimized implementation
  - Optimized version may be platform-specific
- Useful for adding extra VM functionality

Deoptimization

- Sometimes our optimizations are incorrect
  - Faulty assumptions
  - Class loading changes CHA
- If so we need to pull out of compiled code
  - Switch back to bytecode interpretation
  - Expensive operation
- Compiled method becomes one or more frames
  - Undoing the effects of inlining

Cooperative Deoptimization

- Used when an assumption is proven wrong
  - Bad inlining decision
  - Monomorphic to polymorphic switch
- Code was compiled to account for the possibility
  - Extra checks built into machine code
- Deoptimization handler block
  - Hand-coded assembler
  - Compiled code jumps on failed check

Uncooperative Deoptimization

- Not intended by the original compilation
  - Debugger breakpoint or code hot swap
  - Class loading
- Suspending code walks the thread stacks
  - Patches code with jump to deoptimization block
  - Compiled code jumps to the handler block

Safepointing

- Some transformations are disruptive
  - OSR
  - Deoptimization
- We need to VM to be in a consistent state
  - At bytecode boundaries
  - Not running native code
- VM defines safe points when code can change
  - Also useful for garbage collection

Safepointing

- Poll points
  - Interpreter – bytecode boundaries
  - JIT - Method exit and backward branches
- Straightforward in the interpreter
  - Insert a check after every bytecode
- Compiled code uses a poisoned page algorithm
  - JIT inserts memory accesses to a known address
  - VM thread poisons the page (sets as unreadable)
  - All other threads have a memory trap on read