# Status of Java

Fredrik Öhrström
Principal Member of Technical Staff
Oracle

# Status of Java

Fredrik Öhrström
Principal Member of Technical Staff
Oracle

I have worked on the JRockit JVM for the last 6 six years and on the OpenJDK during the last two years.

I am now working in the language team led by the Java Language Architect Brian Goetz.

Though I am currently sidetracked to rewrite the build system for OpenJDK (including making javac multi-core).

# Disclaimer

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

# In the beginning

- In 1974, Donald Knuth wrote a paper with the title *Programming with goto statements*.

# In the beginning

- In 1974, Donald Knuth wrote a paper with the title *Programming with goto statements*.
- "In the late 1960s we witnessed a 'software crisis,' which many people thought paradoxical because programming was supposed to be easy."

# In the beginning

- In 1974, Donald Knuth wrote a paper with the title *Programming with goto statements*.
- "In the late 1960s we witnessed a 'software crisis,' which many people thought paradoxical because programming was supposed to be easy."
- It then takes him approx 60 pages to discuss the problems with for-loops that were relevant at the time.

# In the beginning

- In 1974, Donald Knuth wrote a paper with the title *Programming with goto statements*.
- "In the late 1960s we witnessed a 'software crisis,' which many people thought paradoxical because programming was supposed to be easy."
- It then takes him approx 60 pages to discuss the problems with for-loops that were relevant at the time.
- Today, it is even more complicated!

```
IndexDB db = new IndexDB(5);
int highest_cost = 0;
for (Car c : cars) {
  if (c.manufactured == 2011) {
    if (highest_cost < c.cost_of_repairs * db.index) {
      highest_cost = c.cost_of_repairs * db.index;
    }
  }
}
```

```
IndexDB db = new IndexDB(5);
int highest_cost = 0;
for (Car c : cars) {
  if (c.manufactured == 2011) {
    if (highest_cost < c.cost_of_repairs * db.index) {
      highest_cost = c.cost_of_repairs * db.index;
    }
  }
}
```

- Traversal logic is encoded into bytecode.

```
IndexDB db = new IndexDB(5);
int highest_cost = 0;
for (Car c : cars) {
  if (c.manufactured == 2011) {
    if (highest_cost < c.cost_of_repairs * db.index) {
      highest_cost = c.cost_of_repairs * db.index;
    }
  }
}
```

- Traversal logic is encoded into bytecode.
- Business logic is stateful.

```
IndexDB db = new IndexDB(5);
int highest_cost = 0;
for (Car c : cars) {
  if (c.manufactured == 2011) {
    if (highest_cost < c.cost_of_repairs * db.index) {
      highest_cost = c.cost_of_repairs * db.index;
    }
  }
}
```

- Traversal logic is encoded into bytecode.
- Business logic is stateful.
- Existing collections impose external iteration

```
IndexDB db = new IndexDB(5);
int highest_cost = 0;
for (Car c : cars) {
  if (c.manufactured == 2011) {
    if (highest_cost < c.cost_of_repairs * db.index) {
      highest_cost = c.cost_of_repairs * db.index;
    }
  }
}
```

- Traversal logic is encoded into bytecode.
- Business logic is stateful.
- Existing collections impose external iteration
- For complex datasets the iterator becomes unnecessary complex.

```
IndexDB db = new IndexDB(5);
int highest_cost = 0;
for (Car c : cars) {
  if (c.manufactured == 2011) {
    if (highest_cost < c.cost_of_repairs * db.index) {
      highest_cost = c.cost_of_repairs * db.index;
    }
  }
}
```

- Traversal logic is encoded into bytecode.
- Business logic is stateful.
- Existing collections impose external iteration
- For complex datasets the iterator becomes unnecessary complex.
- But the JVM can do a good job. Lets have a look....

The generated machine code is efficient and minimizes the number of loads from memory. Using for example hoisting of loop invariants and loop peeling.

The generated machine code is efficient and minimizes the number of loads from memory. Using for example hoisting of loop invariants and loop peeling.

But the code is not parallel.

# A stream like approach

```
IndexDB db = new IndexDB(5);
int highest_cost =
 cars.filter(new Predicate<Car>() {
   public boolean op(Car c) {
   return c.manufactured == 2011; }
 }).map(new Extractor<Car,Integer>() {
    public Integer extract(Car c) {
    return c.cost_of_repairs * db.index;  }
  }).max();
```

# A stream like approach

```
IndexDB db = new IndexDB(5);
int highest_cost =
 cars.filter(new Predicate<Car>() {
   public boolean op(Car c) {
   return c.manufactured == 2011; }
 }).map(new Extractor<Car,Integer>() {
    public Integer extract(Car c) {
    return c.cost_of_repairs * db.index;  }
  }).max();
```

- Traversal logic completely inside collection.

# A stream like approach

```
IndexDB db = new IndexDB(5);
int highest_cost =
 cars.filter(new Predicate<Car>() {
   public boolean op(Car c) {
   return c.manufactured == 2011; }
 }).map(new Extractor<Car,Integer>() {
    public Integer extract(Car c) {
    return c.cost_of_repairs * db.index;  }
  }).max();
```

- Traversal logic completely inside collection.
- Stateless!

# Warning! Programmer Syntax Overload!

```
IndexDB db = new IndexDB(5);
int highest_cost =
 cars.filter(new Predicate<Car>() {
   public boolean op(Car c) {
   return c.manufactured == 2011; }
 }).map(new Extractor<Car,Integer>() {
    public Integer extract(Car c) {
    return c.cost_of_repairs * db.index;  }
  }).max();
```

- Traversal logic completely inside collection.
- Stateless!

The status of Java is that it is a lot of work writing software that makes use of all your cores!

The status of Java is that it is a lot of work writing software that makes use of all your cores!

This is *the* major problem that needs to be resolved in the Java language!

The status of Java is that it is a lot of work writing software that makes use of all your cores!

This is *the* major problem that needs to be resolved in the Java language!

A solution must be delivered in jdk8 next year!

# How to improve the Java language

Allow programming patterns that require modeling *code as data* to be convenient and idiomatic in Java. The principal new language features include:

- Lambda expressions (aka “closures” or “anonymous methods”)
- Expanded target typing
- Method and constructor references
- Default methods

# Code as Data?

```
IndexDB db = new IndexDB(5);
int highest_cost =
  cars.filter(c -> c.manufactured == 2011)
  .map(c -> c.cost * db.index)
  .reduce(0, Integer::max);
```

# Code as Data?

```
IndexDB db = new IndexDB(5);
int highest_cost =
  cars.filter(c -> c.manufactured == 2011)
  .map(c -> c.cost * db.index)
  .reduce(0, Integer::max);
```

- An -> identifies a lambda expression

# Code as Data?

```
IndexDB db = new IndexDB(5);
int highest_cost =
  cars.filter(c -> c.manufactured == 2011)
  .map(c -> c.cost * db.index)
  .reduce(0, Integer::max);
```

- An -> identifies a lambda expression
- Target types the parameters

# Code as Data?

```
IndexDB db = new IndexDB(5);
int highest_cost =
  cars.filter(c -> c.manufactured == 2011)
  .map(c -> c.cost * db.index)
  .reduce(0, Integer::max);
```

- An -> identifies a lambda expression
- Target types the parameters
- A :: identifies a method reference.

# Code as Data?

```
IndexDB db = new IndexDB(5);
int highest_cost =
  cars.filter(c -> c.manufactured == 2011)
  .map(c -> c.cost * db.index)
  .reduce(0, Integer::max);
```

# Code as Data?

```
IndexDB db = new IndexDB(5);
int highest_cost =
  cars.filter(c -> c.manufactured == 2011)
  .map(c -> c.cost * db.index)
  .reduce(0, Integer::max);
```

- Code reads like the problem statement: Find the car with the highest indexed cost manufactured in 2011.

# Code as Data?

```
IndexDB db = new IndexDB(5);
int highest_cost =
  cars.filter(c -> c.manufactured == 2011)
  .map(c -> c.cost * db.index)
  .reduce(0, Integer::max);
```

- Code reads like the problem statement: Find the car with the highest indexed cost manufactured in 2011.
- Shorter than nested for loops, and potentially faster because the collection implementation determines how to iterate

# Code as Data?

```
IndexDB db = new IndexDB(5);
int highest_cost =
  cars.filter(c -> c.manufactured == 2011)
  .map(c -> c.cost * db.index)
  .reduce(0, Integer::max);
```

- Code reads like the problem statement: Find the car with the highest indexed cost manufactured in 2011.
- Shorter than nested for loops, and potentially faster because the collection implementation determines how to iterate
- Opportunities for lazy evaluation, but more likely lazy construction of code.

# Lambda Expressions

# Lambda Expressions

- The name comes from the lambda calculus created by Church (1936)

# Lambda Expressions

- The name comes from the lambda calculus created by Church (1936)
- Later explored by Steele and Sussman (1975-1980) in the famous lambda papers.

# Lambda Expressions

- The name comes from the lambda calculus created by Church (1936)
- Later explored by Steele and Sussman (1975-1980) in the famous lambda papers.
- Thus by adding lambda expressions to Java, the status of Java has moved from 1974 (Knuth's structured gotos) to 1975 (Scheme and lexical lambdas).

# Lambda Expressions

- The name comes from the lambda calculus created by Church (1936)
- Later explored by Steele and Sussman (1975-1980) in the famous lambda papers.
- Thus by adding lambda expressions to Java, the status of Java has moved from 1974 (Knuth's structured gotos) to 1975 (Scheme and lexical lambdas).
- Yay! Progress!

# Why were lambda expressions not added earlier to Java?

- After all, Steele who wrote the lambda papers in 1975, worked in the Java team for several years!

# Why were lambda expressions not added earlier to Java?

- After all, Steele who wrote the lambda papers in 1975, worked in the Java team for several years!
- The reason was the design decision that every memory allocation should be visible as a “new” in Java source code. And the reason for this decision was that the hardware of the time simply was not powerful enough.

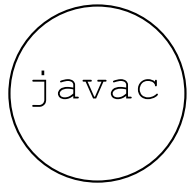
# Lets add this arrow thingy to Java!

# Syntax

```
(int x) -> x+1
(int x, int y) -> x+y+z
(String msg) -> { log(msg); }
```

# Lets add this arrow thingy to Java!

# Lets add this arrow thingy to Java!

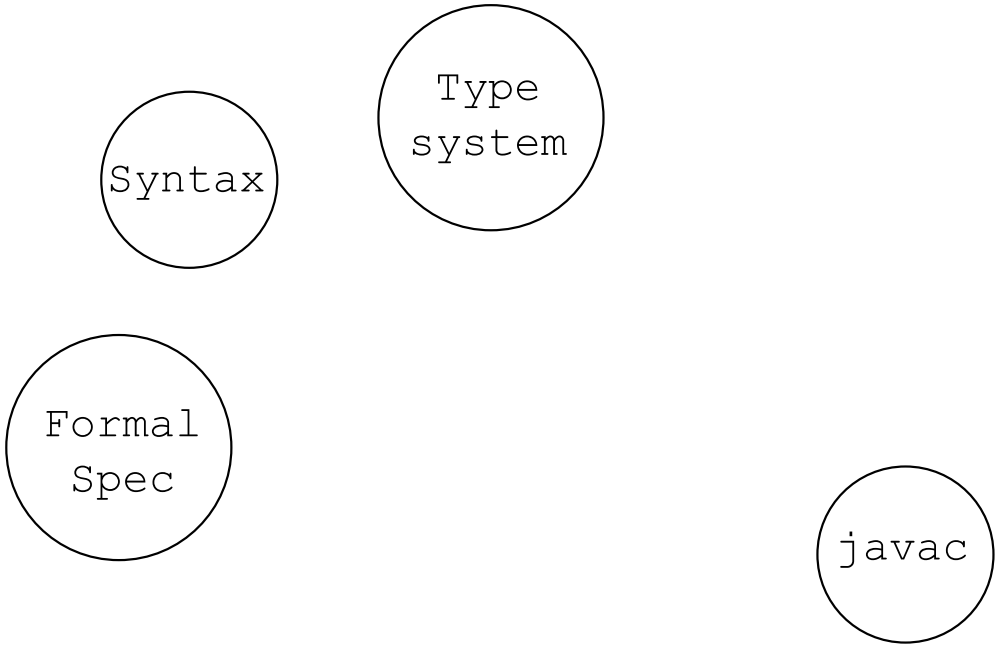
# Lets add this arrow thingy to Java!

# Formal specification

| | |
|---|---|
| Effectively final | `x -> x+z` |
| Void compatible | `x -> { log(42); }` |
| Value compatible | `x -> x+z` |
| | `x -> { return 42; }` |
| Non-local jumps | `break,continue` |
| The meaning of this | `x -> use(this);` |

# Lets add this arrow thingy to Java!

# Lets add this arrow thingy to Java!



# Lambda expressions requires target typing

```
Comparator<T> c = (x,y) -> x.lessThan(y);
FileFilter    f = x -> hasGoodPath(x);
Runnable      r = () -> { doMyStuff(); }
ActionListener a = e -> { doSave(); }
```

# Lambda expressions requires target typing

```
FileFilter      f = Tester::hasGoodPath
Runnable        r = this::doMyStuff;
ActionListener a = this::doSave;
```

# Lambda expressions requires target typing

Unfortunately, no function types!

There will be a lot of standard interfaces that enumerate basic combinations of primitives

Predicate, Mapper, Operator

There is a risk of overproliferation of interfaces.
We could perhaps introduce function types in the future.

# Overloading support requires heuristics!

The type system of Java combined with overloading is so complex that you cannot perform target typing for all cases in a simple way. Eventually, javac has to resort to heuristics to guess what the programmer probably wants.

# Overloading support requires heuristics!

The type system of Java combined with overloading is so complex that you cannot perform target typing for all cases in a simple way. Eventually, javac has to resort to heuristics to guess what the programmer probably wants.

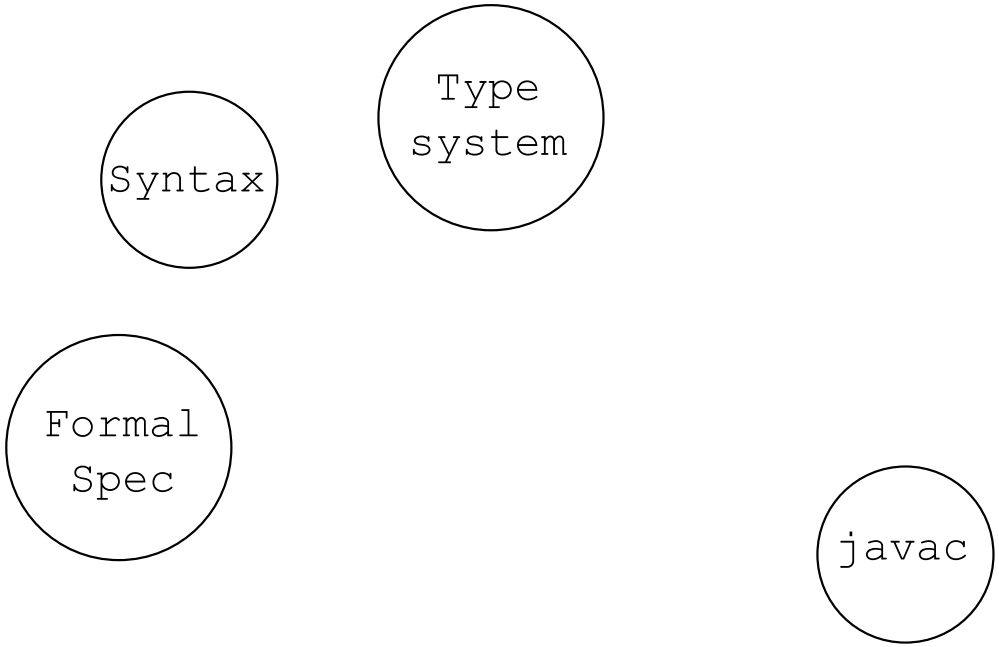
Good for you!

# Type system is undecideable, ouch!

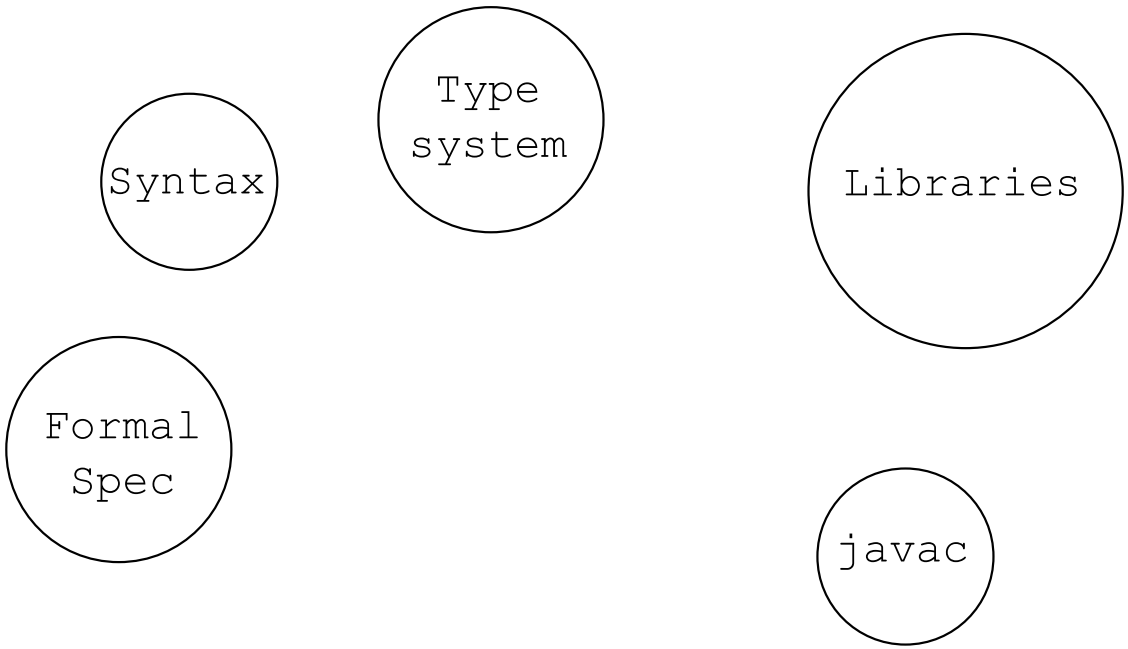
```
class F<T> {}
class C<X extends F<F<? super X>>> {
  C(X x) { F<? super X> f = x; }
}
```

Lets not make it worse....

# Lets add this arrow thingy to Java!



# Lets add this arrow thingy to Java!



# Changing the existing collection classes?

# Changing the existing collection classes?

No! Change the iterators!

```
interface Iterable<T> {
    ...
    Iterable<T> filter(Predicate<T> filter);
    <U> Iterable<U> map(Mapper<T,U> mapper);
    <U> U reduce(U base, Reducer<T,U> reducer);
    void forEach(Block<T> block);
    <U extends Collection<? super T>> U
       intoCollection(U collection);
}
```

```
interface Splittable<T> {
    boolean canSplit();
    Splittable<T> left();
    Splittable<T> right();
    Iterator<T> iterator();
}
```

```
interface Spliterable<T> {
    ...
    Splittable<T> splittable();
    Spliterable<T> filter(Predicate<T> filter);
    <U> Spliterable<U> map(Mapper<T,U> mapper);
    <U> U reduce(U base, Reducer<T,U> reducer);
    void forEach(Block<T> block);
    <U extends Collection<? super T>>
       U intoCollection(U collection);
}
```

```
interface Collection<T> {
   ....
 Spliterable<T> parallel() default Collections::parallel;
}
```

# Code as Data?

```
IndexDB db = new IndexDB(5);
int highest_cost =
  cars.parallel().filter(c -> c.manufactured == 2011)
  .map(c -> c.cost * db.index)
  .reduce(0, Integer::max);
```

# A conversion strategy

First rewrite your code to be stream like.

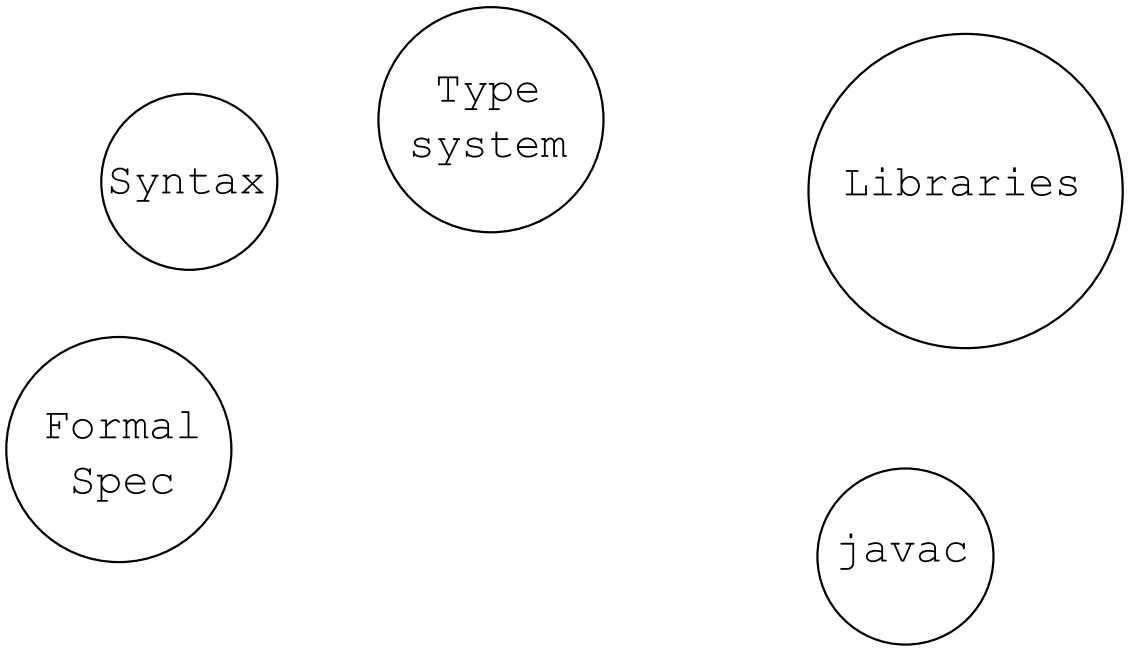
# A conversion strategy

First rewrite your code to be stream like.
Then add .parallel() and see if it gives a speed improvement! :-)

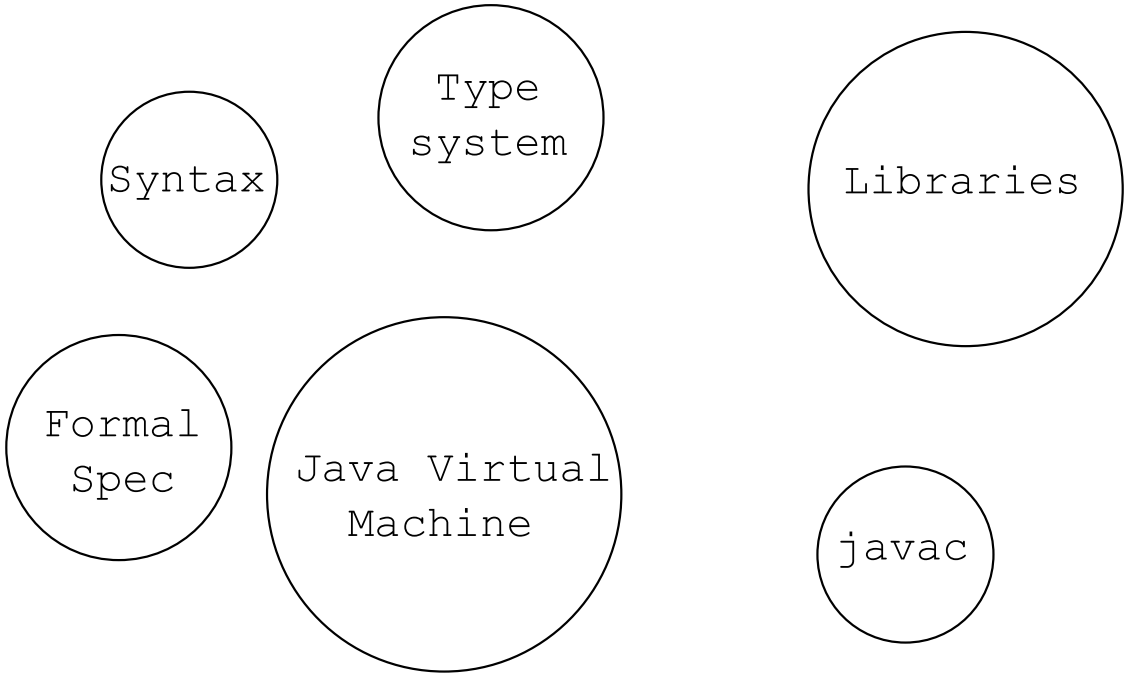
# A conversion strategy

First rewrite your code to be stream like.
Then add .parallel() and see if it gives a speed improvement! :-)
Moving to parallelism must unfortunately be explicit..... too many side effects in normal Java programs.

# Lets add this arrow thingy to Java!



# Lets add this arrow thingy to Java!



# Another optimization example

# Another optimization example

```
static class Coord {
  final int x, y;
  Coord(int xx, int yy) { x=xx; y=yy; }
  Coord add(Coord c) {
    return new Coord(x+c.x,y+c.y);
  }
  double dist(Coord c) {
    int dx = c.x-x;
    int dy = c.y-y;
    return Math.sqrt(dx*dx+dy*dy);
  }
}
```

# Another optimization example

# Another optimization example

```
public static class Matrix {
  int a,b,c,d;
  public Matrix(int aa, int bb, int cc, int dd) {
    a = aa; b = bb; c = cc; d = dd;
  }
  public static Matrix scaler(int k) {
    return new Matrix(k,0,0,k);
  }
  public Matrix mul(int k) {
    return new Matrix(a*k,b*k,c*k,d*k);
  }
  Coord mul(Coord co) {
    return new Coord(co.x*a+co.y*c,co.x*b+co.y*d);
  }
}
```

# Another optimization example

$$\begin{bmatrix} 33 & 0 \\ 0 & 33 \end{bmatrix} \times 2 \times \begin{bmatrix} k \\ k \end{bmatrix} - \begin{bmatrix} 22 \\ 11 \end{bmatrix} = \begin{bmatrix} dx \\ dy \end{bmatrix}$$

$$\texttt{answer} = \sqrt{dx^2 + dy^2}$$

# Another optimization example

$$\begin{bmatrix} 33 & 0 \\ 0 & 33 \end{bmatrix} \times 2 \times \begin{bmatrix} k \\ k \end{bmatrix} - \begin{bmatrix} 22 \\ 11 \end{bmatrix} = \begin{bmatrix} dx \\ dy \end{bmatrix}$$

$$\texttt{answer} = \sqrt{dx^2 + dy^2}$$

```
public static double test(int k) {
 return Matrix.scaler(33).mul(2).mul(new Coord(k,k))
   .dist(new Coord(22,11));
}
```

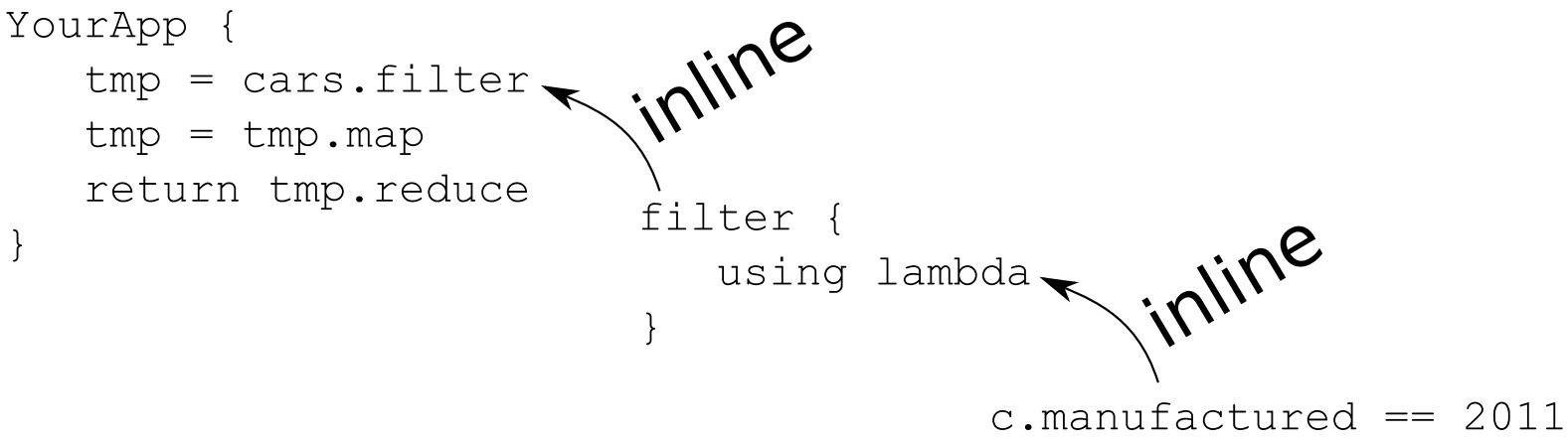
```
x86_sub esp ← esp 4
x86_mov edi ← eax
x86_test *[0xb722f000] eax
x86_imul eax ← eax 66
x86_imul edi ← edi 66
x86_mov ecx ← 22
x86_sub ecx ← ecx eax
x86_mov esi ← 11
x86_sub esi ← esi edi
x86_imul ecx ← ecx ecx
x86_imul esi ← esi esi
x86_add ecx ← ecx esi
x86_cvtsi2sd xmm0 ← ecx
x86_sqrtsd xmm0 ← xmm0
x86_pop ecx esp ← esp
x86_retxmm0 esp
```

Inlining implies great optimization opportunities!

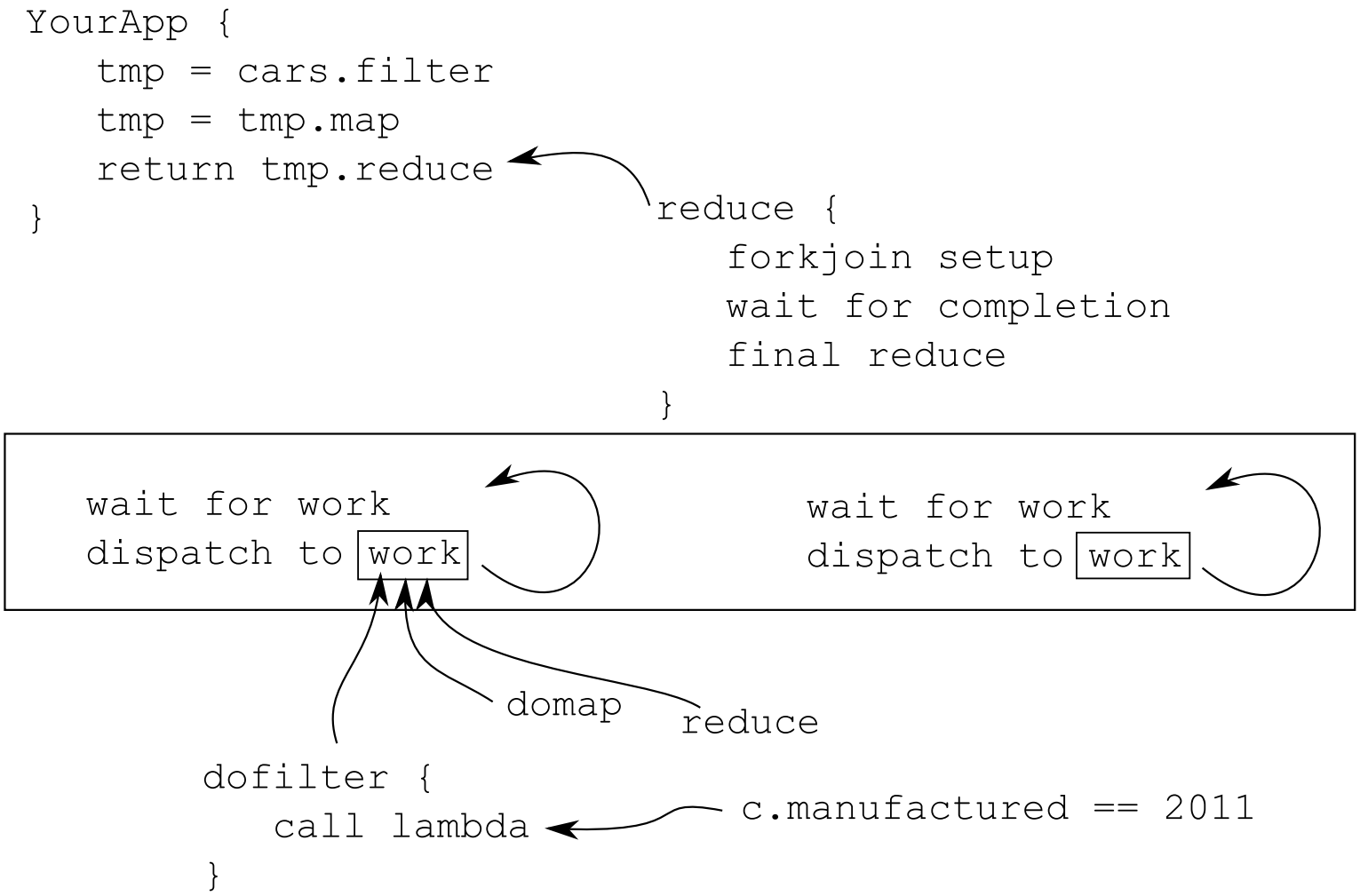
# BUT!

This kind of code optimization is based to the opportunity to do code specialization at the call site!

# Where is the call site in a parallel program?



# Where is the call site in a parallel program?



# Disaster!

We have made code into data, but lost the ability to inline!

# A simple question to calm our nerves...

Why change the meaning of 'this' within a lambda compared to 'this' within an inner class?

```
Runnable r1 = () -> { this.toString(); }
Runnable r2 = new Runnable(){void run(){ this.toString();}}
```

# A simple question to calm our nerves...

Why change the meaning of 'this' within a lambda compared to 'this' within an inner class?

```
Runnable r1 = () -> { this.toString(); }
Runnable r2 = new Runnable(){void run(){ this.toString();}}
```

Well, it makes sense when writing lambdas that look like code. Tennents correspondence principle and its variants. But there is also a deeper technical reason.....

A lambda is an object, but perhaps not in the way you think....

A lambda is an object, but perhaps not in the way you think....

What implementation hides behind the functional interface?

A lambda is an object, but perhaps not in the way you think....

What implementation hides behind the functional interface?


Enter the MethodHandle!

# The MethodHandle

- Is an opaque reference to a method
- Embeds a type to verify that a call is safe at runtime
- Can box and unbox arguments at runtime
- Can be acquired using ldc in the bytecode
- A receiver argument can be bound
- It can be efficiently hidden behind an interface.

# The MethodHandle

- Is an opaque reference to a method
- Embeds a type to verify that a call is safe at runtime
- Can box and unbox arguments at runtime
- Can be acquired using ldc in the bytecode
- A receiver argument can be bound
- It can be efficiently hidden behind an interface.
- I.e. it is not just a function pointer.

# The MethodHandle

- Is an opaque reference to a method
- Embeds a type to verify that a call is safe at runtime
- Can box and unbox arguments at runtime
- Can be acquired using ldc in the bytecode
- A receiver argument can be bound
- It can be efficiently hidden behind an interface.
- I.e. it is not just a function pointer.
- And it can serve as a root for inlining!

# Example 1 (syntax in flux)

```
public class Test1 {
  static volatile MethodHandle mh = Test1::calc(int,Object);

  public static void main(String... args) throws Throwable {
     String a = mh.invoke(1, (Object)"A");
     Object b = mh.invoke(2, "B");
     Object c = mh.invoke((Object)3, 3);
     System.out.println(""+a+","+b+","+c);
   }

   static String calc(int x, Object o) {
     return ""+x+"="+o.hashCode();
   }
}
```

# Example 1 (syntax in flux)

```
public class Test1 {
  static volatile MethodHandle mh = Test1::calc(int,Object);

  public static void main(String... args) throws Throwable {
     String a = mh.invoke(1, (Object)"A");
     Object b = mh.invoke(2, "B");
     Object c = mh.invoke((Object)3, 3);
     System.out.println(""+a+","+b+","+c);
   }

   static String calc(int x, Object o) {
     return ""+x+"="+o.hashCode();
   }
}
```

Will print 1=65,2=66,3=3

# Example 1 (syntax in flux)

```
public class Test1 {
  static volatile MethodHandle mh = Test1::calc(int,Object);

  public static void main(String... args) throws Throwable {
     String a = mh.invoke(1, (Object)"A");
     Object b = mh.invoke(2, "B");
     Object c = mh.invoke((Object)3, 3);
     System.out.println(""+a+","+b+","+c);
   }

   static String calc(int x, Object o) {
     return ""+x+"="+o.hashCode();
   }
}
```

Will print 1=65,2=66,3=3 Weird isn't it?

# Matrix calculation revisited

$$\begin{bmatrix} 33 & 0 \\ 0 & 33 \end{bmatrix} \times 2 \times \begin{bmatrix} k \\ k \end{bmatrix} - \begin{bmatrix} 22 \\ 11 \end{bmatrix} = \begin{bmatrix} dx \\ dy \end{bmatrix}$$

$$\texttt{answer} = \sqrt{dx^2 + dy^2}$$

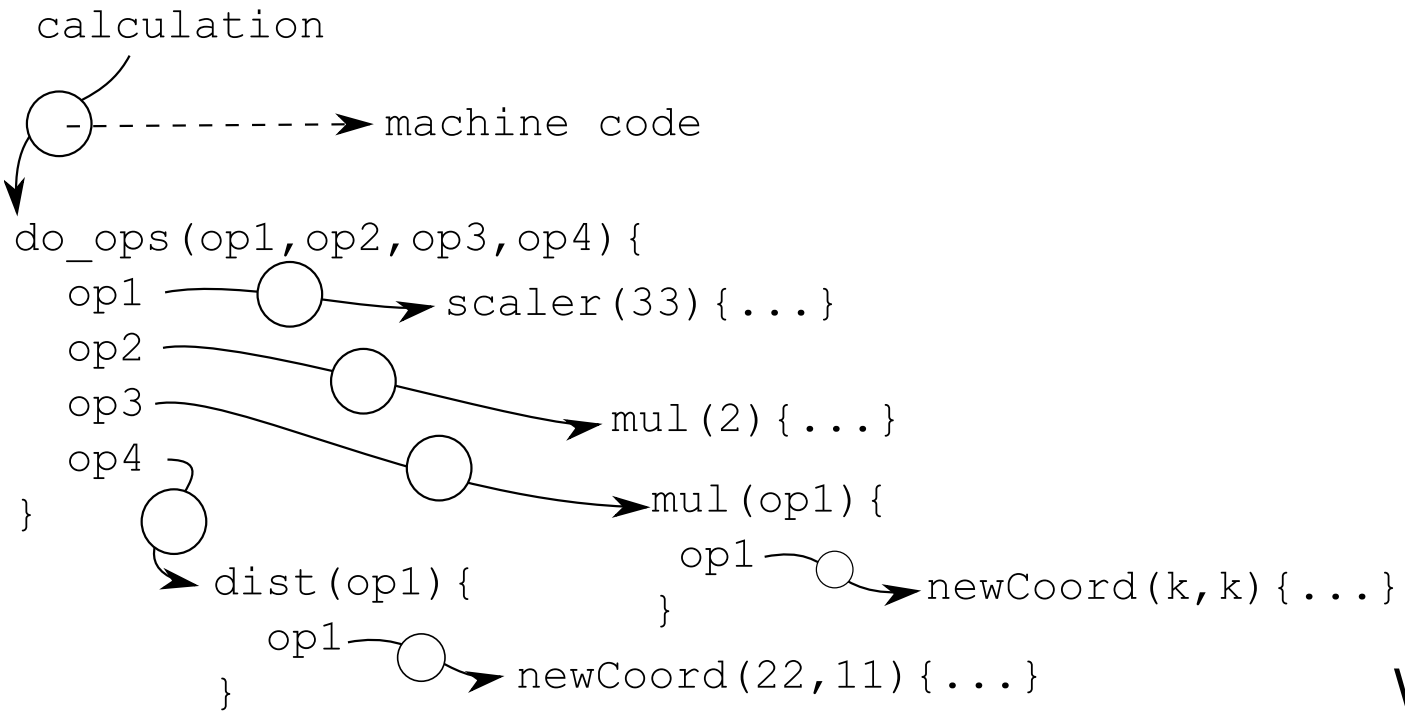
# Matrix calculation revisited

$$\begin{bmatrix} 33 & 0 \\ 0 & 33 \end{bmatrix} \times 2 \times \begin{bmatrix} k \\ k \end{bmatrix} - \begin{bmatrix} 22 \\ 11 \end{bmatrix} = \begin{bmatrix} dx \\ dy \end{bmatrix}$$

$$\texttt{answer} = \sqrt{dx^2 + dy^2}$$

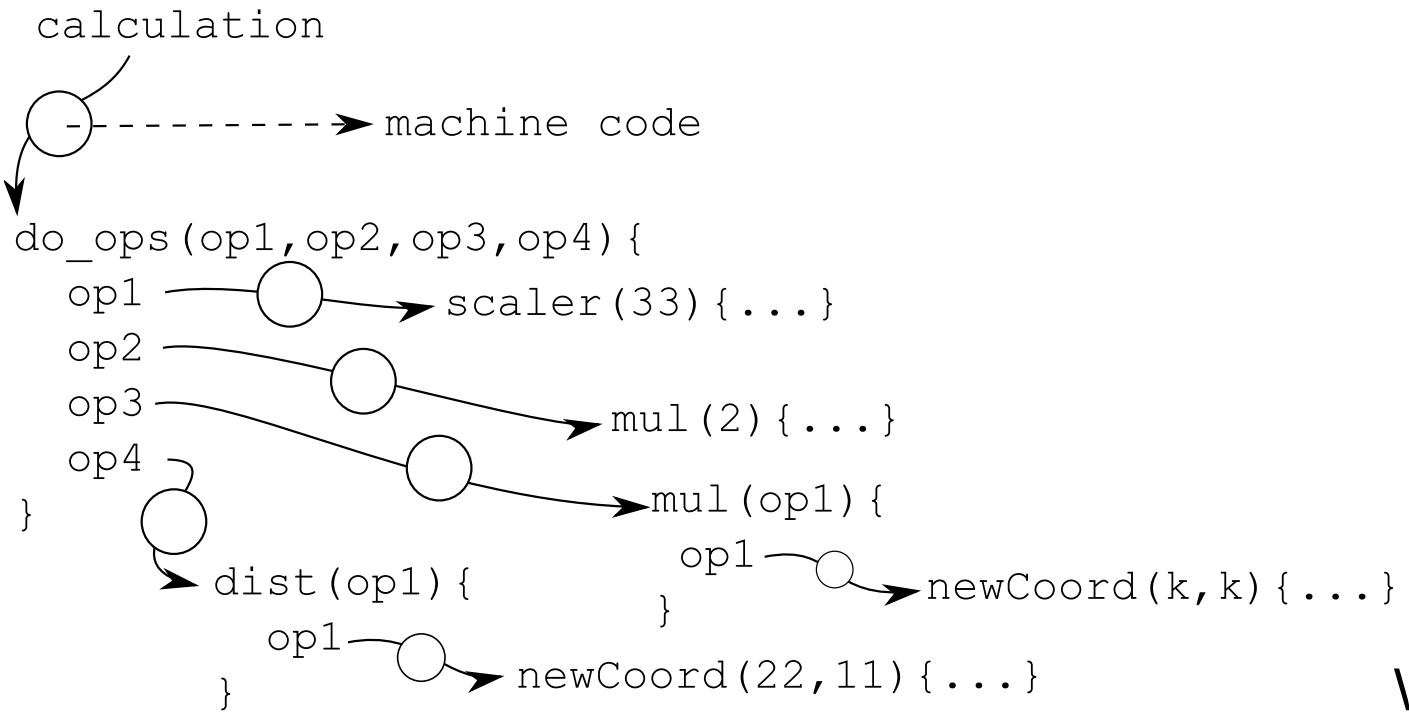
```
public static double test(int k) {
 return Matrix.scaler(33).mul(2).mul(new Coord(k,k))
   .dist(new Coord(22,11));
}
```

# Using MethodHandles




We can create a tree of method handles pointing to small snippets of code. Each methodhandle binds constants like 33,2,22,11,op1,op2,op3,op4 or pass along variables like k.

# Using MethodHandles




We can create a tree of method handles pointing to small snippets of code. Each methodhandle binds constants like 33,2,22,11,op1,op2,op3,op4 or pass along variables like k. Code has now become data that can become optimized code again!

# Lets go back to the serial loop

# Lets go back to the serial loop

```
static class Car {
    public int manufactured;
    public int cost_of_repairs;
}
static class IndexDB {
    public IndexDB(int i) { index = 125*i; }
    public int index;
}
public static int report() {
    Car[] cars = new Car[1];
    cars[0] = new Car();
    cars[0].cost_of_repairs = 17;
    cars[0].manufactured = 2011;
    return calc(cars);
}
```

# Lets go back to the serial loop

```
public static int calc(Car[] cars) {
  int highest_cost = 0;
  IndexDB db = new IndexDB(7);
  for (Car c : cars) {
    if (c.manufactured == 2011) {
   if (highest_cost < c.cost_of_repairs * db.index) {
       highest_cost = c.cost_of_repairs * db.index;
     }
    }
  }
  return highest_cost;
}
```

JRockit optimized the code a single machine code instruction!

```
mov eax, 14875
```

JRockit optimized the code a single machine code instruction!

```
mov eax, 14875
```

Imagine that the spliterator splits out regions whose boundaries are encoded in the method handle tree encapsulating the work packet.

I.e. not only can we specialize the code based on the call site, but also on the data being calculated!

JRockit optimized the code a single machine code instruction!

```
mov eax, 14875
```

Imagine that the spliterator splits out regions whose boundaries are encoded in the method handle tree encapsulating the work packet. I.e. not only can we specialize the code based on the call site, but also on the data being calculated! Hotspot might do the same in the future. We are working hard on moving JRockit functionality into the OpenJDK.

# Code is Data!

# Code is Data!

But now we need to avoid reconstructing the same code over and over again!

```
IndexDB db = new IndexDB(5);
int highest_cost =
cars.parallel().filter(c -> c.manufactured == 2011)
.map(c -> c.cost * db.index)
.reduce(0, Integer::max);
```

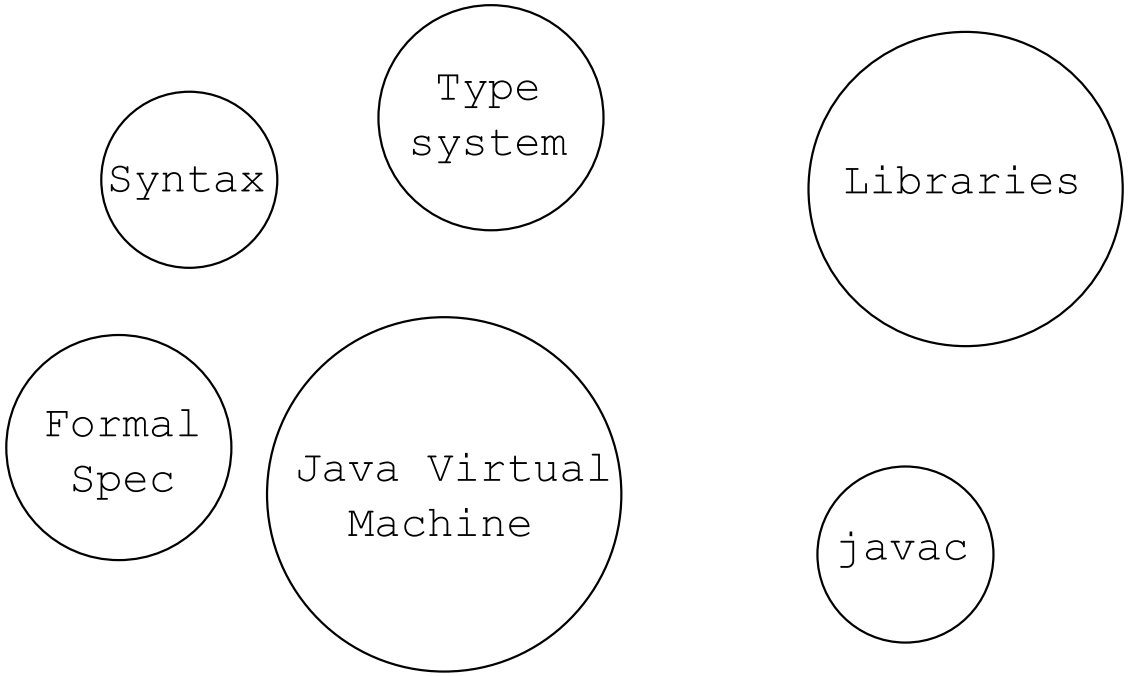
# Code is Data!

We use invokedynamic to store constructed code.

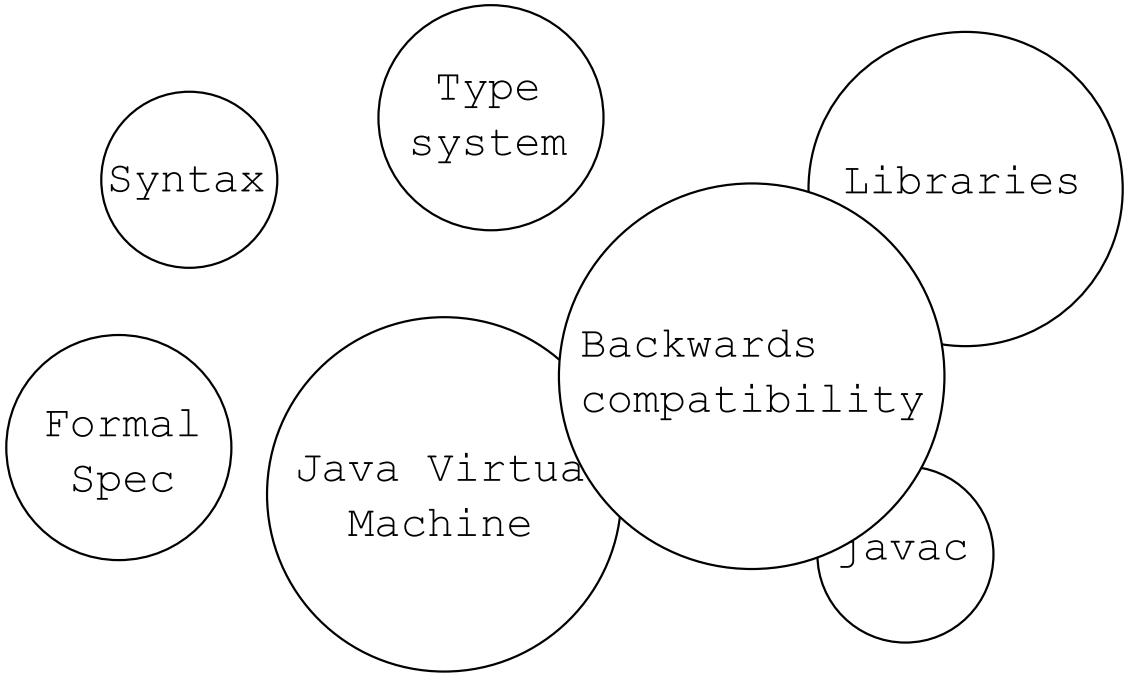
```
IndexDB db = new IndexDB(5);
Spliterable sp1 = cars.parallel();
Predicate p = INDY bootstrap:
                c -> c.manufactured == 2011
Spliterable sp2 = sp1.filter(p);
Mapper m = INDY bootstrap:
             c -> c.cost * db.index
Spliterable sp3 = sp2.map(m);
Reducer r = INDY bootstrap:
              Integer::max
int highest_cost = sp3.reduce(0, r);
              INDY inside reduce
              stores the method handle tree.
```

Phew!

# Lets add this arrow thingy to Java!



# Lets add this arrow thingy to Java!



# The wet blanket of backwards compatibility

You can use lambdas or method references everywhere in your old code where you use inner anonymous classes.

For example:
`save_button.onAction(this::save)`

It hides hacks in the bytecode, ie the bridge methods used to fake co/contra variant methods.

It puts parts of the type system in the JVM to deal with these problems.
(Not enough to make it undecideable though.)

# Summary

Serial vs Parallel (multi core)

# Summary

Serial vs Parallel (multi core)
Syntax vs Inlineability (streams and spliterables)

# Summary

Serial vs Parallel (multi core)
Syntax vs Inlineability (streams and spliterables)
Code vs Data (methodhandles)

# Summary

Serial vs Parallel (multi core)
Syntax vs Inlineability (streams and spliterables)
Code vs Data (methodhandles)
Reuse compiled code vs Reuse constructed code (invokedynamic)

# Status of Java

The Java language is getting a lambda operator and a Mount Everest of technical improvements to assist this seemingly simple operator.

# Status of Java

The Java language is getting a lambda operator and a Mount Everest of technical improvements to assist this seemingly simple operator.

- We have a schedule to keep!
  jdk8 release date September 2013.

# Status of Java

The Java language is getting a lambda operator and a Mount Everest of technical improvements to assist this seemingly simple operator.

- We have a schedule to keep!
  jdk8 release date September 2013.
- We already have a working implementation that you can try out!

# Status of Java

The Java language is getting a lambda operator and a Mount Everest of technical improvements to assist this seemingly simple operator.

- We have a schedule to keep!
  jdk8 release date September 2013.
- We already have a working implementation that you can try out!
- Oracle's motto is: Keep Java vibrant.

# Status of Java

The Java language is getting a lambda operator and a Mount Everest of technical improvements to assist this seemingly simple operator.

- We have a schedule to keep!
  jdk8 release date September 2013.
- We already have a working implementation that you can try out!
- Oracle's motto is: Keep Java vibrant.

Thank you!

fredrik.ohrstrom@oracle.com