Lambda Expressions in Java SE 8

Weiqi Gao
St. Louis Java Users Group
January 10, 2013*

*information may be out-of-date past this date

About The Presenter

- Java developer since 1998
- Principal Software Engineer at OCI
- Co-author of Pro JavaFX 2
- Steering committee of St. Louis JUG
- http://weiqigao.blogspot.com/ and @weiqigao

Project Lambda

- Home: http://openjdk.org/projects/lambda
- Developer Preview: http://jdk8.java.net/lambda
  - for Windows, Linux, Solaris, Mac OS X
- JSR 335: Lambda Expressions for Java http://jcp.org/en/jsr/detail?id=335

Presentations, Etc.

- Strange Loop 2012: Dan Smith Talk (slides)
- JavaOne 2012: Brian Goetz (in Technical Keynote)
- Devoxx 2012: Brian Goetz, Language / Library / VM Co-Evolution in Java SE 8
- Maurice Neftalin's Lambda FAQ

Lambda Expressions

- ```
  EventQueue.invokeLater(new Runnable() {
      @Override
      public void run() {
          swingMain(args);
      }
  });
  ```
- becomes
- ```
  EventQueue.invokeLater(() -> {
      swingMain(args);
  });
  ```

Lambda Expressions

- `() -> {}`
- `x -> x`
- `x -> { int y = x; return y; }`
- `(x) -> { System.out.println(this); }`
-
```
(String x) -> {
  throw new RuntimeException();
}
```

Lambda Expressions

- Can't do the following
  - -> {} // must use () -> {} for empty args
  - (x, int y) -> x + y // types are all or none
  - (int x) throws Exception -> 42 // can't throw

Where Is It Allowed?

- RHS of variable declarations and assignments
- Return statements;
- Method/constructor arguments
- Array initializers, Casts, p ? x : y
- No syntax for immediate invocation à la
  - ((lambda (x) x) 10) ; scheme
  - (function() {})(); // JavaScript

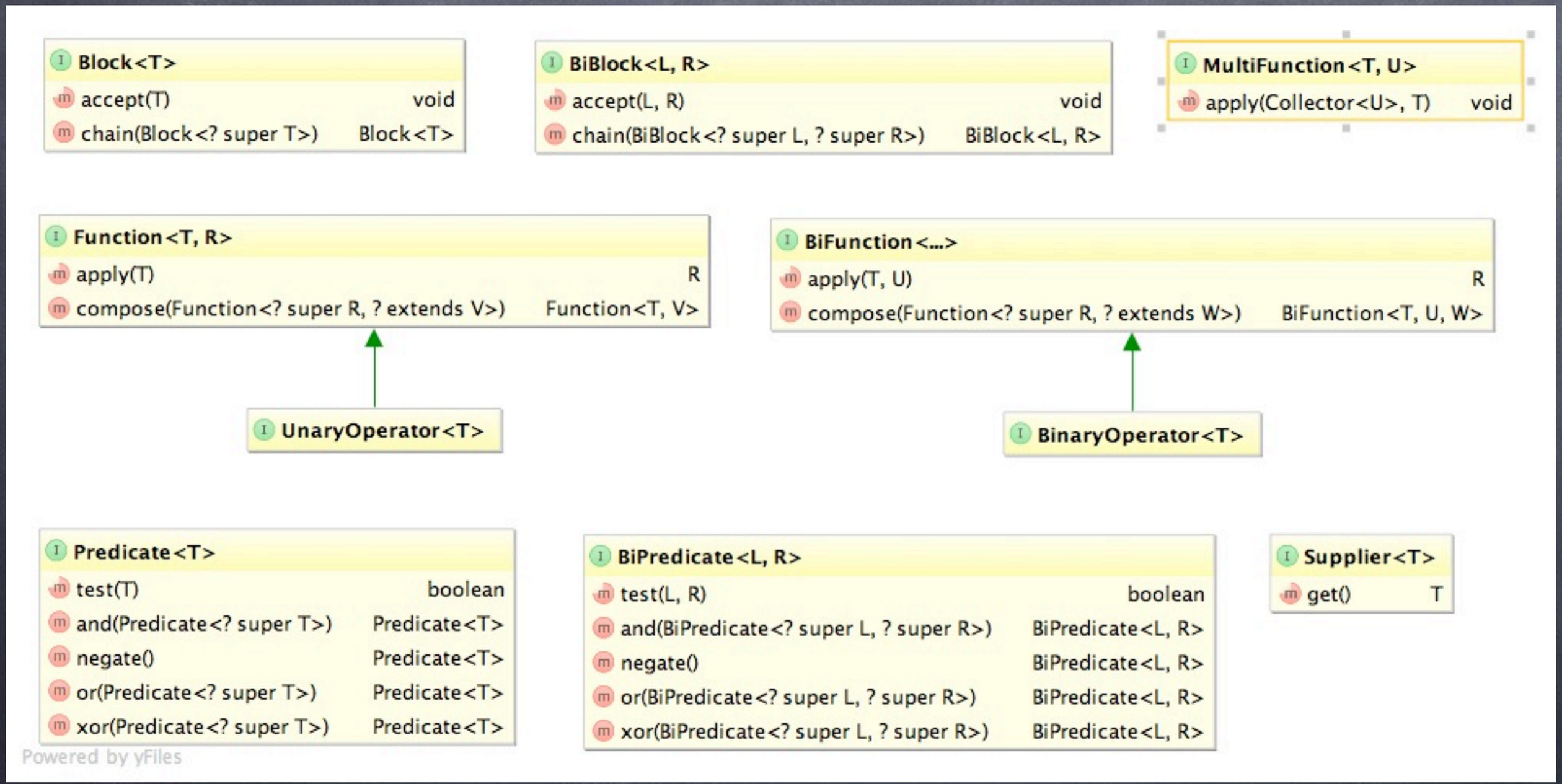
What Are Their Types?

- Receiver must be of "Functional Interface" type
- An interface with one abstract method
  - concrete (and static) methods are allowed in interfaces in Java 8
- System.out.println(x -> x); won't compile
- Nor would System.out.println("" + x -> x);

Functional Interfaces

- Many interfaces in the JDK have one method
  - Runnable, Callable, ActionListener
  - Many recent libraries are designed with it in mind, e.g., JavaFX's EventListener
  - But not java.awt.event.MouseListener
- A set of brand new ones in java.util.function that covers most use cases

java.util.function



More Examples

- ```
  Block<String> block = x -> {
      System.out.println("x = " + x);
  };
  block.accept("Hello");
  ```
- ```
  BiBlock<String, String> biBlock = (x, y) -> {
      System.out.println("x = "+x +", y = "+y);
  };
  biBlock.accept("Hello", "World");
  ```

More Examples

- Function<Long, Double> f = x -> 1.0d / x;
  System.out.println("f.apply(3L) = " +
  f.apply(3L));
- BiFunction<String, String, String> bi =
  (x, y) -> x + " " + y;
  System.out.println("biF.apply(\"Hello\",
  \"World\") = " + biF.apply("Hello", "World"));

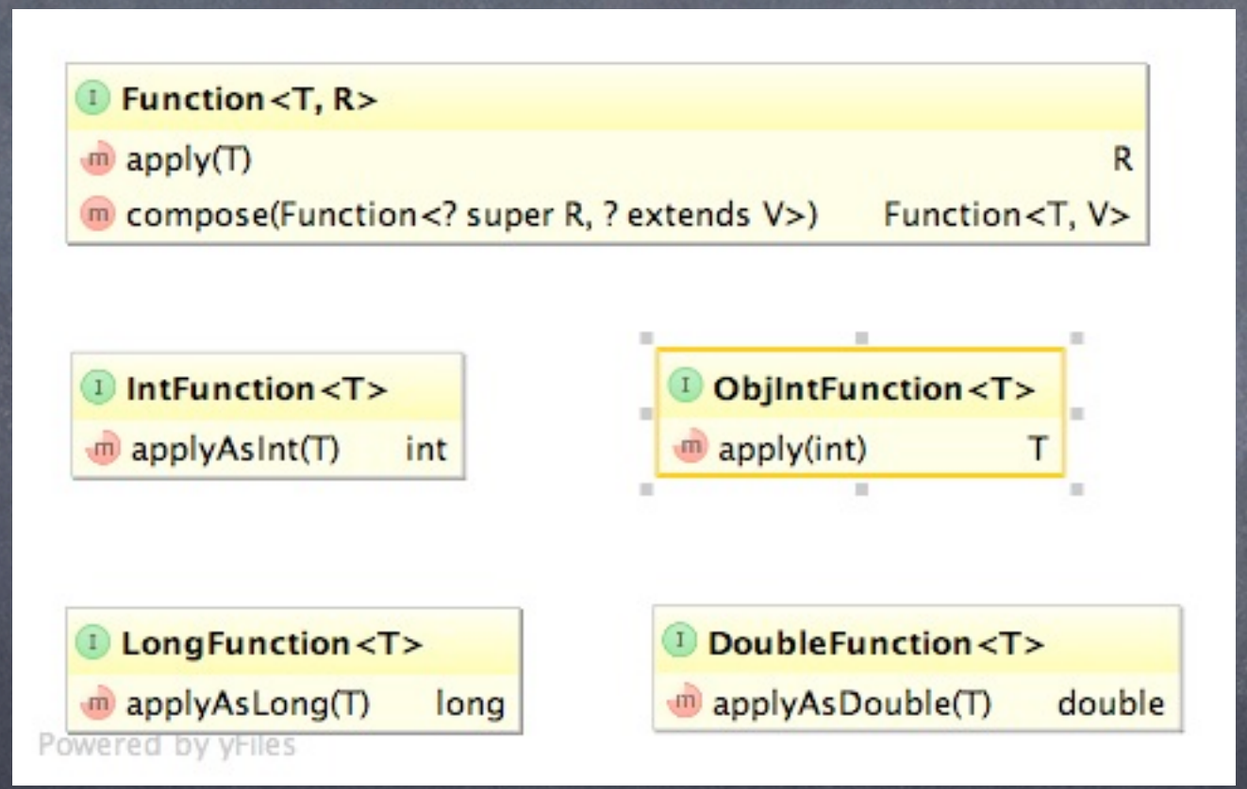
More Examples

- UnaryOperator<String> uOp = s ->
  s.toUpperCase();
  System.out.println("uOp.apply(\"hello\") = " +
  uOp.apply("hello"));
- BinaryOperator<Integer> bOp = (i, j) -> i * j;
  System.out.println("bOp.apply(3, 7) = " +
  bOp.apply(3, 7));

More Examples

- Predicate<String> p = x -> x.contains("a");
  System.out.println("p.test(\"banana\") = " +
  p.test("banana"));
- BiPredicate<Integer, Integer> biP =
  (i, j) -> i < j;
  System.out.println("biP.test(10, 4) = " +
  biP.test(10, 4));

For Primitive Types



More Examples

- IntFunction<String> intF = x-> x.length();
  System.out.println(
  "intF.applyAsInt(\"Hello\") = " +
  intF.applyAsInt("Hello"));
- Avoids unnecessary boxing and unboxing
- Other specialized types also exist
  - IntBlock, LongPredicate, etc.

Names Don't Matter

- Only the signature is considered when an lambda expression is assigned to a functional interface typed variable
- Neither the name of the functional interface, nor the name of the method name matter
- It's kind of a structural type, but only when the lambda expression is evaluated
- Afterwards, you are dealing with normal Java variables

What does this mean?

- The this variable inside a lambda expression refers to the surrounding object, not the object of the functional interface type that the lambda expression evaluates to
- Similarly for the super variable
- Runnable r = new Runnable() {

```
    void run() {
        System.out.println(this);
    }
};  // can't be turned into lambda expression
```

Effectively Final Variables

- A variable that's not final but are never assigned after initialization
- Both final and effectively final variables can be referenced in lambda expressions
- This also applies to anonymous inner classes

Example

```
void f(int i) {
  double d = 3.1415926;
  Block<String> b = x -> {
    System.out.println(x+": i = "+i+", d = "+d);
  };
  b.accept("Hello");
  new Runnable() {
    @Override
    public void run() {
      System.out.println("i = "+i+", d = "+d);
    }
  }.run();
}
```

Method/Constructor References

- System::getProperty
- String::length
- “abc”::length
- Date::new
- Saves you from having to write
  x -> System.getProperty(x)

Examples

- Function<String, String> m = System::getProperty;
- BiFunction<String, String, String> m2 = System::getProperty;
- IntFunction<String> m3 = String::length;
- IntSupplier m4 = "abc"::length;
- Function<Long, Date> m5 = Date::new;

Type Inference, Etc.

- The compiler can make calculations about types involved in lambda expressions
- Therefore in certain circumstances you don't need to specify the parameter types
- The rules used are fairly involved
- Especially when generics are involved

Can We Do a Fibonacci?

```
public class Fibonacci {
    static interface F {
        int invoke(int i);
    }
    private static F fib = x ->
            x == 0 ? 0 :
            x == 1 ? 1 :
            fib.invoke(x - 1) + fib.invoke(x - 2);
    public static void main(String[] args) {
        System.out.println(fib.invoke(10));
    }
}
```

Default Methods

- Java interfaces can now have concrete methods (with a body)
- They are called default methods, and must have the "default" modifier
- They are introduced so that interfaces in libraries can be extended without breaking old code
- Collections framework is extended with functional programming idioms and methods

Other Changes To Interfaces

- Static methods are allowed
- package can be used as modifier to methods
- Fields are still not allowed
- Now that interfaces may contain implementations, we need to be careful about diamond inheritance scenarios

Example

```
interface A {
    void f();
    static void g() {}
    default void h() {}
}
class B implements A {
    @Override
    public void f() {}
}
```

Iterable gains forEach

```
public interface Iterable<T> {
    Iterator<T> iterator();
    public default void forEach(Block<? super
T> block) {
        for (T t : this) {
            block.accept(t);
        }
    }
}
```

Example

```
public class InternalIteration {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("Hello");
        list.add("World");
        list.forEach(s -> {
            System.out.println("s = " + s);
        });
    }
}
```

Collections and Streams

- Collection gains default methods stream() and parallelStream() that return Streams
- Stream is a new interface that represent a collection in mid-processing
- Streams methods may return Streams for chained processing
- Streams may be infinite

Stream<T>

filter(Predicate<? super T>)	Stream<T>
map(Function<? super T, ? extends R>)	Stream<R>
map(IntFunction<? super T>)	IntStream
mapMulti(MultiFunction<? super T, R>)	Stream<R>
uniqueElements()	Stream<T>
sorted(Comparator<? super T>)	Stream<T>
forEach(Block<? super T>)	void
tee(Block<? super T>)	Stream<T>
limit(long)	Stream<T>
substream(long)	Stream<T>
substream(long, long)	Stream<T>
into(A)	A
toArray()	Object[]
reduce(T, BinaryOperator<T>)	T
reduce(BinaryOperator<T>)	Optional<T>
reduce(U, BiFunction<U, T, U>, BinaryOperator<U>)	U
accumulate(Accumulator<T, R>)	R
accumulate(Supplier<R>, BiBlock<R, T>, BiBlock<R, R>)	R
accumulateConcurrent(ConcurrentTabulator<T, R>)	R
max(Comparator<? super T>)	Optional<T>
min(Comparator<? super T>)	Optional<T>
anyMatch(Predicate<? super T>)	boolean
allMatch(Predicate<? super T>)	boolean
noneMatch(Predicate<? super T>)	boolean
findFirst()	Optional<T>
findAny()	Optional<T>
sequential()	Stream<T>
parallel()	Stream<T>
unordered()	Stream<T>

Powered by yFiles

Example

```
public class FunctionalProgramming {
  public static void main(String[] args) {
    List<String> list = Arrays.asList("Hello",
      "World");
    Optional<StringHolder> result =
      list.stream()
        .filter(s -> s.length() == 5)
        .map(s -> new StringHolder(s))
        .reduce((x, y) -> {
          return new StringHolder(x.value +
            " " + y.value);
        });
```

```
    if (result.isPresent()) {
      System.out.println("result.get().value);
    } else {
      System.out.println(result.toString());
    }
  }
}
```

Parallel Processing

- When you request a parallelStream() from a collection, the resulting Stream is parallel capable
- It will perform certain Stream operations in parallel on multi/many-core machines using the fork/join framework
- It does this with the help of a Spliterator
  - Iterator returns 1 and keeps n - 1
  - Spliterator returns k and keeps n - k

Example

```
int[] ints = new int[1024];
for (int i = 0; i < 1024; i++) { ints[i] = i; }
int sum = Arrays.parallelStream(ints)
        .filter(i -> {
          System.out.println(
            Thread.currentThread().getName());
          return true;
        })
        .sum();
```

Laziness Is a Virtual

- If you omit the final sum() call in the last example, nothing will happen
- The intermediate methods on Stream (those that returns another Stream) are mostly lazy when they can

That's All Good, But Can You Do a Y-combinator?

- (define Y
  (lambda (f)
    ((lambda (x)
      (f (x x)))
     (lambda (x)
      (f (x x))))))
- I haven't tried
- Left as an exercise for the reader

Summary

- Lambda expressions
- Functional interfaces
- Target typing
- Default Methods
- Library enhancements (Collections, Streams)
- Parallel processing -> Free lunch is back!