
JavaOne™

Jump-Starting Lambda

Stuart Marks @stuartmarks
Mike Duigou @mjduigou

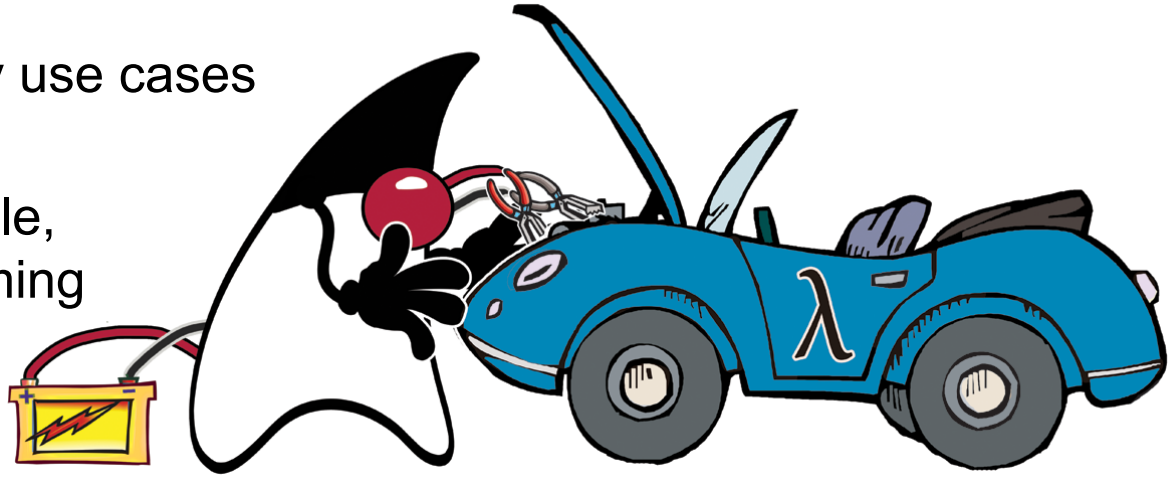
Oracle JDK Core Libraries Team

What is Lambda?

- Essentially an anonymous function
  - allows one to treat code as data
  - provides parameterization of ***behavior*** as opposed to ***values*** or ***types***
  - provides “closures” (but we’re not exploring closure properties here)
- Combined with JVM and class library changes, supports
  - high productivity, flexible, “fluent” style of programming
  - explicit but unobtrusive parallelism
  - useful for simple, everyday programming
  - also heavy (parallel) lifting

Presentation Overview

- Exploring programming techniques leading toward Lambda
- Little emphasis on syntax and language definition
- Examples driven by use cases
- Illustrate lambda's usefulness for simple, everyday programming tasks



Example Overview

- Person object
- Using a collection of Person objects (e.g. a List)
- Selecting some of these Person objects
  - e.g., by age or sex
- Operations using Person objects
  - e.g., making an automated phone call (“robocall”)

Canonical Example Object

```
class Person {
    int getAge();
    Sex getSex();
    PhoneNumber getPhoneNumber();
    EmailAddr getEmailAddr();
    PostalAddr getPostalAddr();
    /* ... */
}
```


Robocall Every Person

```
void robocallEveryPerson() {
    List<Person> list = gatherPersons();
    for (Person p : list) {
        PhoneNumber num = p.getPhoneNumber();
        robocall(num);
    }
}
```


Some Use Cases

- Select only eligible drivers
  - in California, must be 16 years or older
- Select only eligible voters
  - in the U.S., must be 18 years or older
- Select only persons of legal drinking age
  - in California, must be 21 years or older

Robocall Eligible Drivers

```
void robocallEligibleDrivers() {
    List<Person> list = gatherPersons();
    for (Person p : list) {
        if (p.getAge() >= 16) {
            PhoneNumber num = p.getPhoneNumber();
            robocall(num);
        }
    }
}
```


Robocall Eligible Voters

```
void robocallEligibleVoters() {
    List<Person> list = gatherPersons();
    for (Person p : list) {
        if (p.getAge() >= 18) {
            PhoneNumber num = p.getPhoneNumber();
            robocall(num);
        }
    }
}
```


Robocall Persons of Legal Drinking Age

```
void robocallLegalDrinkers() {
    List<Person> list = gatherPersons();
    for (Person p : list) {
        if (p.getAge() >= 21) {
            PhoneNumber num = p.getPhoneNumber();
            robocall(num);
        }
    }
}
```


Duplicated code!
Notice a pattern?

Solution: Add a Parameter!

```
void robocallPersonsOlderThan(int age) {
    List<Person> list = gatherPersons();
    for (Person p : list) {
        if (p.getAge() >= age) {
            PhoneNumber num = p.getPhoneNumber();
            robocall(num);
        }
    }
}
```


Using Parameters

```
void robocallDrivers() {
    robocallPersonsOlderThan(16);
}

void robocallVoters() {
    robocallPersonsOlderThan(18);
}

void robocallDrinkers() {
    robocallPersonsOlderThan(21);
}
```


Additional Use Case

- Commercial pilots have mandatory retirement age of 65
- Previously we parameterized a ***value***
- Now, we want to parameterize ***less-than*** or ***greater-than***
  - Add a boolean?

Boolean Parameter Encodes Less-or-Greater

```
void robocallPersonsWithAgeLimit(int age, boolean lessThan)
{
    List<Person> list = gatherPersons();
    for (Person p : list) {
        if ( lessThan && (p.getAge() < age) ||
             !lessThan && (p.getAge() >= age)) {
            PhoneNumber num = p.getPhoneNumber();
            robocall(num);
        }
    }
}
```


Using the Boolean Parameter

```
void robocallEligibleDrivers()
    { robocallPersonsWithAgeLimit(16, false); }

void robocallEligibleVoters()
    { robocallPersonsWithAgeLimit(18, false); }

void robocallLegalDrinkers()
    { robocallPersonsWithAgeLimit(21, false); }

void robocallEligiblePilots()
    { robocallPersonsWithAgeLimit(65, true); }
```


Let's face it, this sucks

Corrected Use Case

- Commercial pilots also have a ***minimum*** age of 23
  - as well as a mandatory retirement age of 65
- Use an age range instead...

Parameters for Age Range

```
void robocallPersonsInAgeRange(int low, int high) {
    List<Person> list = gatherPersons();
    for (Person p : list) {
        if (low <= p.getAge() && p.getAge() < high) {
            PhoneNumber num = p.getPhoneNumber();
            robocall(num);
        }
    }
}
```


Using Age Range


```
void robocallEligibleDrivers()
    { robocallPersonsInAgeRange(16, MAX); }

void robocallEligibleVoters()
    { robocallPersonsInAgeRange(18, MAX); }

void robocallLegalDrinkers()
    { robocallPersonsInAgeRange(21, MAX); }

void robocallEligiblePilots()
    { robocallPersonsInAgeRange(23, 65);  }
```

Additional Use Case

- United States Selective Service (national service program)
  - Age range 18 to 25 (inclusive)

```
void robocallSelectiveService() {
    robocallPersonsInAgeRange(18, 26);
}
```


Corrected Use Case

- United States Selective Service
  - Age range 18 to 25 (inclusive), ***men only***
- Change to:

```
void robocallSelectiveService() {
    robocallPersonsInAgeRange(18, 26, MALE);
}
```

```
enum Sex { MALE, FEMALE }
```

- Works, but what about queries that don't care about sex?
  - Add `DONT_CARE` to enum Sex? Mixes data with query value.
  - Use null? (Yes, enums can be null; weird special case.)

Parameterized Computation

- Value parameterization is useful, but only to a certain point
  - Problem: meta-information is communicated “in-band”
  - In these cases we need a value that means “don’t care”
  - Sometimes special values can be found that work:
    - `0, -1, Integer.MAX_VALUE, null`
  - If not, add more boolean, enum parameters
- Just age and sex, but already we sort of need a query language
- A giant leap: parameterize ***behavior*** instead of values
  - Make a method’s parameter be a ***function*** instead of a ***value***

Functions: Parameters and Return Values

(int, int) → int	example: addition
String → int	example: length
Person → boolean	example: several, let's revisit

Examples of Person → boolean Functions

```
boolean checkIfPersonIsEligibleToDrive(Person p) {
    return p.getAge() >= 16;
}

boolean checkIfPersonIsEligibleToVote(Person p) {
    return p.getAge() >= 18;
}

boolean checkIfPersonIsLegalToDrink(Person p) {
    return p.getAge() >= 21;
}
```


Examples of Person → boolean Functions

```
boolean checkIfPersonIsEligiblePilot(Person p) {
    return p.getAge >= 23 && p.getAge() < 65;
}

boolean checkIfPersonEligibleForSelectiveService(Person p)
{
    return p.getSex() == MALE &&
           p.getAge() >= 18 &&
           p.getAge() <= 25;
}
```


How To Pass a Function in Java?

- Use anonymous inner classes:

```
new Thread(
    new Runnable() {
        public void run() {
            System.out.println("I'm another thread!");
        }
    }
).start();
```


What is the Type of a Function in Java?

( ) → void

```
interface Runnable {
    public void run();
}
```

empty argument list

*This is a **functional interface** — an interface with a single method **

* To be precise, we should say *single abstract method,* since in Java 8 it is now possible for interfaces to have implementations via the *default method* feature.

Predicate: Function Returning Boolean

Person → boolean

```
interface PersonPredicate {
    boolean testPerson(Person p);
}
```


A Generalized Predicate

T → boolean

```
interface Predicate<T> {
    boolean test(T t);
}
```


Using a Function Passed as a Parameter

```
void robocallMatchingPersons(Predicate<Person> pred) {
    List<Person> list = gatherPersons();
    for (Person p : list) {
        if (pred.test(p)) {
            PhoneNumber num = p.getPhoneNumber();
            robocall(num);
        }
    }
}
```


Passing a Function as a Parameter

```
void robocallEligibleDrivers() {
    robocallMatchingPersons(
        new Predicate<Person>() {
            public boolean test(Person p) {
                return p.getAge() >= 16;
            }
        });
}
```


Passing a Function as a Parameter

```
void robocallEligibleVoters()  {
    robocallMatchingPersons(
        new Predicate<Person>() {
            public boolean test(Person p) {
                return p.getAge() >= 18;
            }
        });
}
```


Passing a Function as a Parameter

```
void robocallLegalDrinkers()   {
    robocallMatchingPersons(
        new Predicate<Person>() {
            public boolean test(Person p) {
                return p.getAge() >= 21;
            }
        });
}
```


Passing a Function as a Parameter

```
void robocallEligiblePilots()  {
    robocallMatchingPersons(
        new Predicate<Person>() {
            public boolean test(Person p) {
                return p.getAge() >= 23 &&
                       p.getAge() < 65;
            }
        });
}
```


Passing a Function as a Parameter

```
void robocallSelectiveServiceCandidates()  {
    robocallMatchingPersons(
        new Predicate<Person>() {
            public boolean test(Person p) {
                return p.getSex() == MALE &&
                       p.getAge() >= 18 &&
                       p.getAge() <= 25;
            }
        });
}
```


Powerful, but verbose, and very, very painful

The Significant Part

```
void robocallEligibleDrivers() {
    robocallMatchingPersons(
        new Predicate<Person>() {
            public boolean test(Person p) {
                return p.getAge() >= 16;
            }
        });
}
```


The Boilerplate

```
void robocallEligibleDrivers() {
    robocallMatchingPersons(
        new Predicate<Person>() {
            public boolean test(Person p) {
                return p.getAge() >= 16;
            }
        });
}
```


Erase the Boilerplate

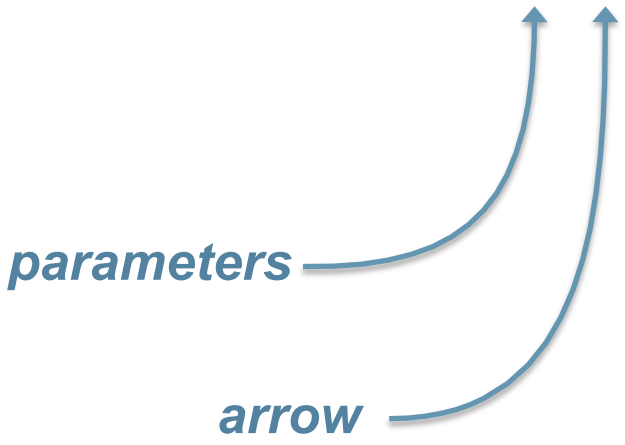
```
void robocallEligibleDrivers() {
    robocallMatchingPersons(

                                        p
                    -> p.getAge() >= 16

        );
}
```


Our First Lambda Expression!

```
void robocallEligibleDrivers() {
    robocallMatchingPersons(p -> p.getAge() >= 16);
}
```

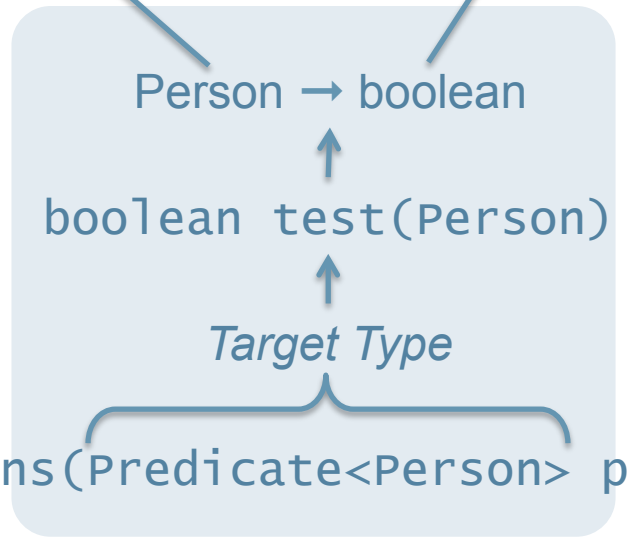


Lambda expressions are converted to instances of functional interfaces

Our First Lambda Expression!

```
void robocallEligibleDrivers() {
    robocallMatchingPersons(p -> p.getAge() >= 16);
}
```

Type Inference:




```
void robocallMatchingPersons(Predicate<Person> pred) {
    ...
}
```

Rewrites Using Lambda

```
void robocallEligibleDrivers() {
    robocallMatchingPersons(p -> p.getAge() >= 16);
}

void robocallEligibleVoters()  {
    robocallMatchingPersons(p -> p.getAge() >= 18);
}

void robocallLegalDrinkers()   {
    robocallMatchingPersons(p -> p.getAge() >= 21);
}
```


Rewrites Using Lambda

```
void robocallEligiblePilots()  {
    robocallMatchingPersons(
        p -> p.getAge() >= 23 && p.getAge() < 65);
}

void robocallSelectiveServiceCandidates()  {
    robocallMatchingPersons(p -> p.getSex() == MALE &&
                                 p.getAge() >= 18 &&
                                 p.getAge() <= 25);
}
```


Where Else Can We Use Lambda?

```
void robocallMatchingPersons(Predicate<Person> pred) {
    List<Person> list = gatherPersons();
    for (Person p : list) {
        if (pred.test(p)) {
            PhoneNumber num = p.getPhoneNumber();
            robocall(num);
        }
    }
}
```


Where Else Can We Use Lambda?

```
void txtmsgMatchingPersons(Predicate<Person> pred) {
    List<Person> list = gatherPersons();
    for (Person p : list) {
        if (pred.test(p)) {
            PhoneNumber num = p.getPhoneNumber();
            txtmsg(num);
        }
    }
}
```


Functional Interface: Block

PhoneNumber → void

T → void

```
interface Block<T> {
    void apply(T t);
}
```


Extract & Parameterize Block Function

```
void processMatchingPersons(Predicate<Person> pred,
                            Block<PhoneNumber> block) {
    List<Person> list = gatherPersons();
    for (Person p : list) {
        if (pred.test(p)) {
            PhoneNumber num = p.getPhoneNumber();
            block.apply(num);
        }
    }
}
```


Examples of Block

```
processMatchingPersons(p -> p.getAge() >= 16,
                       num -> { robocall(num); });

processMatchingPersons(p -> p.getAge() >= 18,
                       num -> { txtmsg(num); });
```

These are "statement lambdas" as opposed to "expression lambdas"

What Else Can We Extract?

```
void processMatchingPersons(Predicate<Person> pred,
                            Block<PhoneNumber> block) {
    List<Person> list = gatherPersons();
    for (Person p : list) {
        if (pred.test(p)) {
            PhoneNumber num = p.getPhoneNumber();
            block.apply(num);
        }
    }
}
```


Functional Interface: Mapper

Person → PhoneNumber

T → U

```
interface Mapper<T,U> {
    U map(T t);
}
```


Extract & Parameterize Mapper Function

```
void processMatchingPersons(Predicate<Person> pred,
                            Mapper<Person,PhoneNumber> mapper,
                            Block<PhoneNumber> block)
{
    List<Person> list = gatherPersons();
    for (Person p : list) {
        if (pred.test(p)) {
            PhoneNumber num = mapper.map(p);
            block.apply(num);
        }
    }
}
```

Examples of Mapper

```
processMatchingPersons(p -> p.getAge() >= 16,
                       p -> p.getHomePhoneNumber(),
                       num -> { robocall(num); });

processMatchingPersons(p -> p.getAge() >= 18,
                       p -> p.getMobilePhoneNumber(),
                       num -> { txtmsg(num); });
```


Refactoring & Cleanup

```
void processMatchingPersons(Predicate<Person> pred,
                            Mapper<Person,PhoneNumber> mapper,
                            Block<PhoneNumber> block)
{
    List<Person> list = gatherPersons();
    for (Person p : list) {
        if (pred.test(p)) {
            PhoneNumber num = mapper.map(p);
            block.apply(num);
        }
    }
}
```


Refactoring & Cleanup

```
void processMatchingPersons(List<Person> list,
                            Predicate<Person> pred,
                            Mapper<Person,PhoneNumber> mapper,
                            Block<PhoneNumber> block)
{
    for (Person p : list) {
        if (pred.test(p)) {
            PhoneNumber num = mapper.map(p);
            block.apply(num);
        }
    }
}
```


Refactoring & Cleanup

```
void processMatchingPersons(Iterable<Person> source,
                            Predicate<Person> pred,
                            Mapper<Person,PhoneNumber> mapper,
                            Block<PhoneNumber> block)
{
    for (Person p : source) {
        if (pred.test(p)) {
            PhoneNumber num = mapper.map(p);
            block.apply(num);
        }
    }
}
```


Usage After Refactoring

```
// robocall eligible drivers
processMatchingPersons(gatherPersons(),
                       p -> p.getAge() >= 16,
                       p -> p.getHomePhoneNumber(),
                       num -> { robocall(num); });

// text-message eligible voters
processMatchingPersons(gatherPersons(),
                       p -> p.getAge() >= 18,
                       p -> p.getMobilePhoneNumber(),
                       num -> { txtmsg(num); });
```

What Does processMatchingPersons Really Do?

```
void processMatchingPersons(Iterable<Person> source,
                            Predicate<Person> pred,
                            Mapper<Person,PhoneNumber> mapper,
                            Block<PhoneNumber> block)
{
    for (Person p : source) {
        if (pred.test(p)) {
            PhoneNumber num = mapper.map(p);
            block.apply(num);
        }
    }
}
```

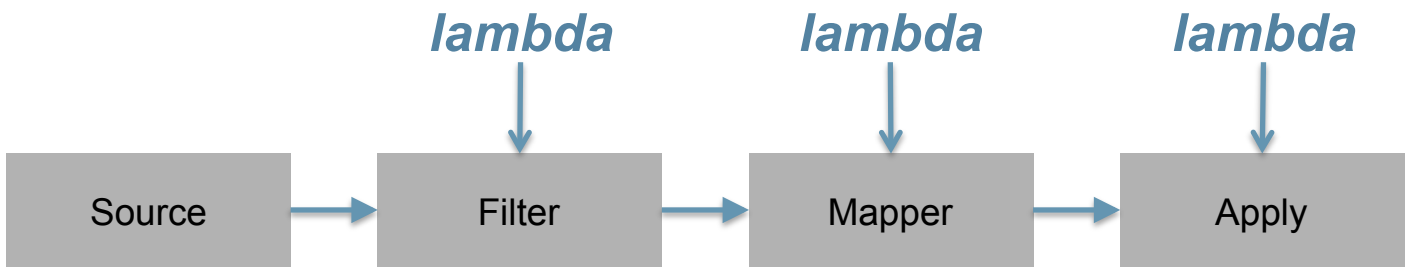

What Does processMatchingPersons Really Do?

```
<< source provides Person objects >> {
    << filter each Person through a predicate >> {
        << map each Person to a PhoneNumber >>
        << apply a function to each PhoneNumber >>
    }
}
```


A General Form of processMatchingPersons

```
<< source provides elements >> {
    << filter each through a predicate >> {
        << map each value to another value >>
        << apply a function to each value >>
    }
}
```


A General Form of processMatchingPersons



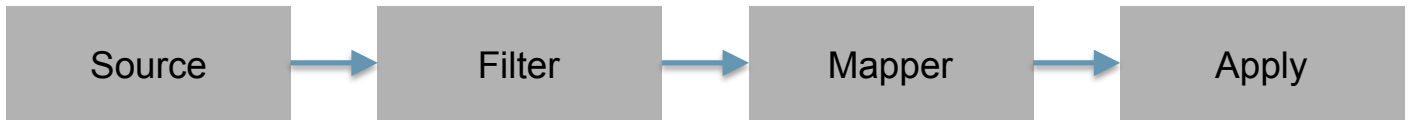
OK great, but can we really write the code this way?

Building a Pipeline for Robocalling

```
void robocallEligibleDrivers() {
    gatherPersons()
```

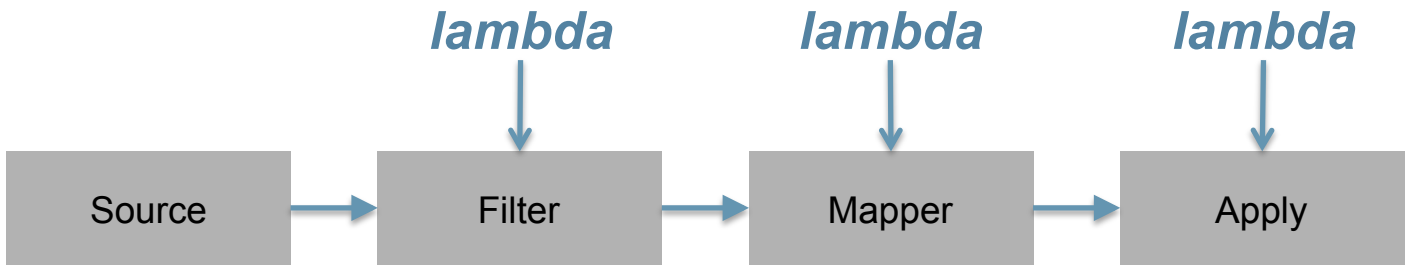

Building a Pipeline for Robocalling

```
void robocallEligibleDrivers() {
    gatherPersons().stream()
        .filter(...)
        .map(...)
        .forEach(...);
}
```



Building a Pipeline

```
void robocallEligibleDrivers() {
    gatherPersons().stream()
        .filter(p -> p.getAge() >= 16)
        .map(p -> p.getHomePhoneNumber())
        .forEach(num -> { robocall(num); });
}
```



Changing the Pipeline

```
void robocallEligibleDrivers() {
    gatherPersons().stream()
        .filter(p -> p.getAge() >= 16)
        .map(p -> p.getHomePhoneNumber())
        .filter(num -> !num.isOnDoNotCallList())
        .forEach(num -> { robocall(num); });
}
```

Changing the Pipeline

```
void textMessageEligibleDrivers() {
    gatherPersons().stream()
        .filter(p -> p.getAge() >= 16)
        .map(p -> p.getMobilePhoneNumber())
        .filter(num -> !num.isOnDoNotCallList())
        .forEach(num -> { txtmsg(num); });
}
```

Changing the Pipeline

```
void spamEligibleDrinkers() {
    gatherPersons().stream()
        .filter(p -> p.getAge() >= 21)
        .map(p -> p.getEmailAddr())
        .forEach(addr -> { sendEmail(addr); });
}
```


Changing the Pipeline

```
void bulkMailEligiblePilots() {
    gatherPersons().stream()
        .filter(p -> p.getAge() >= 23 && p.getAge() < 65)
        .map(p -> p.getPostalAddr())
        .forEach(addr -> { sendBulkMail(addr); });
}
```


Changing the Pipeline

```
void recruitSelectiveServiceCandidates() {
    gatherPersons().stream()
        .filter(p -> p.getSex() == MALE &&
                p.getAge() >= 18 &&
                p.getAge() <= 25)
        .map(p -> p.getHomeAddr())
        .forEach(addr -> { sendArmyRecruiter(addr); });
}
```

How To Do Stuff in Parallel?

```
void robocallEligibleDrivers() {
    gatherPersons().parallelStream()
        .filter(p -> p.getAge() >= 16)
        .map(p -> p.getHomePhoneNumber())
        .forEach(num -> { robocall(num); });
}
```


Functional Interfaces in java.util.functions

`BinaryOperator<T>`	(T, T) → T
`Block<T>`	T → void
`Combiner<T,U,V>`	(T, U) → V
`Factory<T>`	() → T
`FlatMapper<T,R>`	T → R*
`Mapper<T,U>`	T → U
`Predicate<T>`	T → boolean
`UnaryOperator<T>`	T → T

Stream Operations

java.util.stream.Stream

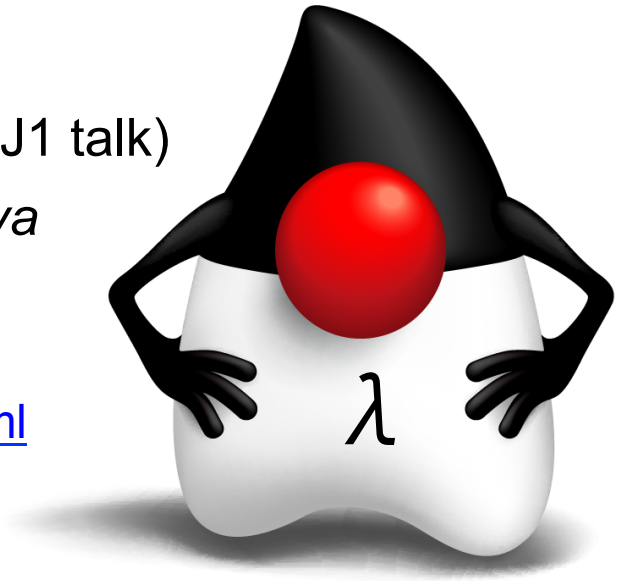
- filter(pred)
- map(mapper)
- forEach(block)
- flatMap(flat-mapper)
- tee(block)
- uniqueElements()
- sorted(comparator)
- into(collection)
- toArray(array-factory)
- any / all / noneMatch(pred)
- findFirst / Any(pred)
- cumulate(binop)
- reduce(binop) / reduce(base, binop)
- fold(base-factory, reducer, combiner)
- groupBy(mapper)
- reduceBy(mapper, base-factory, reducer)

Summary

- Lambdas are anonymous functions
  - They enable parameterization of behavior
- Lambda expressions are converted to functional interfaces
  - A functional interface is an interface with a single (abstract) method
- Two kinds: expression lambdas and statement lambdas
- Compiler type inference reduces boilerplate
- Lambda + new library APIs enable writing programs that are:
  - ***powerful, fluid, expressive, parallel***

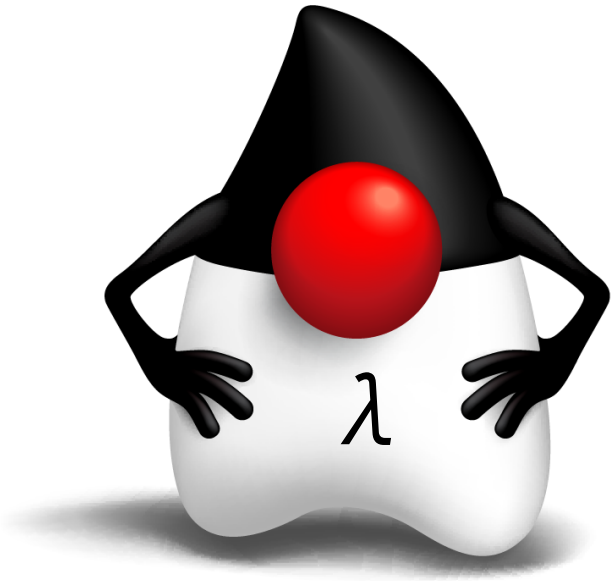
Related Talks – All Brian Goetz, All The Time

- *The Road To Lambda* (earlier J1 talk)
- *Lambda: A Peek Under the Hood* (earlier J1 talk)
- *Implementing Lambda Expressions in Java*
  - JVM Language Summit 2012
  - http://www.oracle.com/technetwork/java/javase/community/jvmls2012-1840099.html
- *Language / Library / VM Co-evolution in Java SE 8,* Devoxx 2011
  - https://blogs.oracle.com/briangoetz/entry/slides_from_devoxx_talk_on

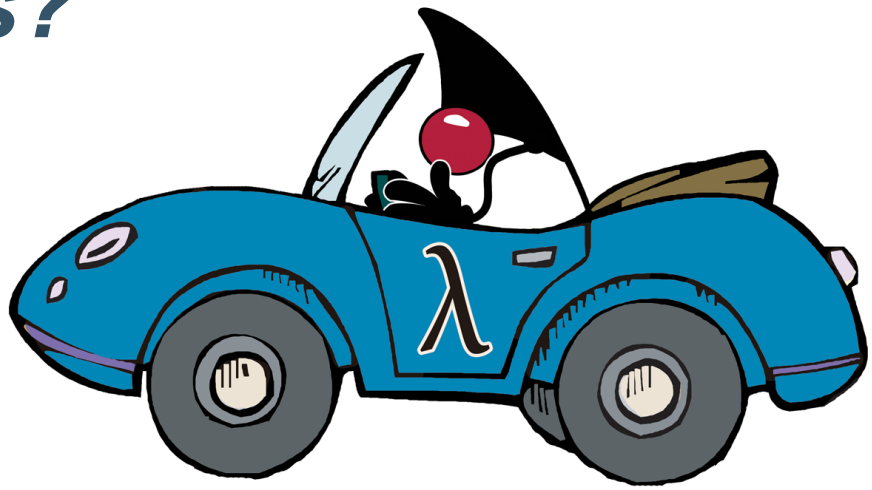


Links

- Lambda project page:
  http://openjdk.java.net/projects/lambda/
- Source code:
  http://hg.openjdk.java.net/lambda/lambda
- Binaries:
  http://jdk8.java.net/lambda/
- FAQ (Maurice Naftalin):
  http://www.lambdafaq.org/



Any Questions?



JavaOne™ ORACLE®

The preceding 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.


MAKE THE
FUTURE
JAVA
JavaOne
ORACLE