
Java™

Introduction to Lambda

Stuart W. Marks
Principal Member of Technical Staff
Oracle JDK Core Libraries Team

Twitter: @stuartmarks

ORACLE

What is a Lambda?

- A lambda is a function.
- A function is a computation that takes parameters and returns a value.
- Until Java 8, functions could only be implemented using methods.
- A lambda enables functions to be passed around or stored like data.

Example: A Simple Person Object

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

enum Sex { FEMALE, MALE }
```

Let's run through some examples.

Example: Robocall Eligible Drivers

```
class MyApplication {

    List<Person> list = ...

    void robocallEligibleDrivers() {
        ...
    }
}
```

Robocall: automated telephone calling for marketing or survey purposes.

Example: Robocall Eligible Drivers

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

In California, 16 is the legal driving age.

What if we want other selection criteria?

Solution: Parameterize Method!

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

robocallMatchingPersons(16);
robocallMatchingPersons(18);
robocallMatchingPersons(21);
```

*Parameterization: separation of **common** elements from **specific** or **varying** elements*

New Requirement

- United States Selective Service (national service program)
  - Age range 18 through 25, ***men only,*** must register
- How to retrofit our method?

```
void robocallSelectiveService() {
    robocallMatchingPersons(MALE, 18, 25);
}
```

Robocall with Sex and Age Range

```
void robocallMatchingPersons(Sex sex, int low, int high) {
    for (Person p : list) {
        if (p.getSex() == sex &&
                low <= p.getAge() && p.getAge() <= high) {
            PhoneNumber num = p.getPhoneNumber();
            robocall(num);
        }
    }
}
```

Problem: Retrofitting Old Cases

```
// Selective Service: males 18-25
void robocallSelectiveService() {
    robocallMatchingPersons(MALE, 18, 25);
}

// drivers: anyone 16 or older
void robocallEligibleDrivers() {
    robocallMatchingPersons( ??? , 16, MAX_VALUE );
}
```

Fix Attempt #1 – Add DONT_CARE

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

void robocallMatchingPersons(Sex sex, int low, int high) {
    for (Person p : list) {
        if ((sex == DONT_CARE || p.getSex() == sex) &&
                low <= p.getAge() && p.getAge() < high) {
            PhoneNumber num = p.getPhoneNumber();
            robocall(num);
        }
    }
}
```

Most people consider this to be a bad idea.

Fix Attempt #2 – Use Null

```
void robocallMatchingPersons(Sex sex, int low, int high) {
    for (Person p : list) {
        if ((sex == null || p.getSex() == sex) &&
                low <= p.getAge() && p.getAge() < high) {
            PhoneNumber num = p.getPhoneNumber();
            robocall(num);
        }
    }
}
```

Retrofit Using Fix Attempt #2

```
// Selective Service: males 18-25
void robocallSelectiveService() {
    robocallMatchingPersons(MALE, 18, 25);
}

// drivers: anyone 16 or older
void robocallEligibleDrivers() {
    robocallMatchingPersons(null, 16, MAX_VALUE);
}
```

But what if Person.getSex() == null means "sex unknown" and you want to search for people whose sex is unknown?

Fix Attempt #3 – Add Boolean

```
void robocallMatchingPersons(Sex sex, boolean matchSex,
                             int low, int high) {
    for (Person p : list) {
        if ((!matchSex || p.getSex() == sex) &&
                low <= p.getAge() && p.getAge() < high) {
            PhoneNumber num = p.getPhoneNumber();
            robocall(num);
        }
    }
}
```

Retrofit Using Fix Attempt #3

```
// Selective Service: males 18-25
void robocallSelectiveService() {
    robocallMatchingPersons(MALE, true, 18, 25);
}

// drivers: anyone 16 or older
void robocallEligibleDrivers() {
    robocallSelectedPersons(null, false, 16, MAX_VALUE);
}
```

You still have to put something, even though it's ignored!

JavaOne ORACLE

What Just Happened?

- Even a simple query got complicated really fast.
- We're only querying for age and sex but we're already designing a mini-query language in our parameter list.
  - Problem: query information mixed with meta-information
  - 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`
  - But it's not always possible.
- After a while, *value parameterization* runs out of power.

Rethink This Like An API Designer

- The ***caller*** wants to:
  - specify selection criteria in a flexible way
  - not be limited to a fixed parameter list
- The ***library*** wants to:
  - reuse traversal and operational logic
- The ***API*** between the caller and the library needs to support this
  - but what data types and values can it use?

What The Caller Wants

```
robocallMatchingPersons( Given a Person, using characteristics of that Person,
                         decide whether that Person should be robocalled. );
```

Given a Person, using characteristics of that Person, decide whether that Person should be robocalled.

What The Library Wants

```
void robocallMatchingPersons( Given a Person, using characteristics of that Person, decide whether that Person should be robocalled. ){
    for (Person p : list) {
        if ( Given a Person, using characteristics of that Person, decide whether that Person should be robocalled. ){
            PhoneNumber num = p.getPhoneNumber();
            robocall(num);
        }
    }
}
```

What is this thing? It's a function!

Given a Person, using characteristics of that Person,

decide whether that Person should be robocalled.

What is this thing? It's a function!

Given a Person, using characteristics of that Person,

The function's parameter is a Person

decide whether that Person should be robocalled.

The function's result is a boolean

How To Represent a Function in Java?

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

Collections.sort(people, new Comparator<Person>() {
    public int compare(Person x, Person y) {
        return x.getLastName().compareTo(y.getLastName());
    }
});
```

Prior to Java 8, we use anonymous inner classes to represent functions

How To Represent a Function Type in Java?

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

The types used for anonymous inner classes are often single-method interfaces.

```
interface Comparator<T> {
    public int compare(T x, T y);
}
```

*Such interfaces are used for representing functions, so we call them **functional interfaces**.*

Function Type of Decide-to-Robocall-Person

```
/**
 * Given a Person, using characteristics of that Person,
 * decide whether that Person should be robocalled.
 */

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

A "predicate" is a function that returns a boolean.

*PersonPredicate is a function that goes **from** Person **to** boolean.*

Apply This To Our Library API

```
void robocallMatchingPersons(PersonPredicate pred) {
    for (Person p : list) {
        if (pred.test(p)) {
            PhoneNumber num = p.getPhoneNumber();
            robocall(num);
        }
    }
}
```

Hey, this is really cool!

Now Over to the Caller's Side

```
// Selective Service: males 18-25

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

Well, maybe not so cool after all.

... And Drivers

```
// drivers: anyone 16 or older

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

Parameterized Behavior

- We've made a giant leap: parameterize ***behavior*** instead of values
- This is great for the library and the API!
- This is not so great for callers of the library.
- But it's such a powerful technique that people do it anyway.

OH: "The pain of anonymous inner classes makes us roll our eyes in the back of our heads every day."

Let's Look At The Code Again

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

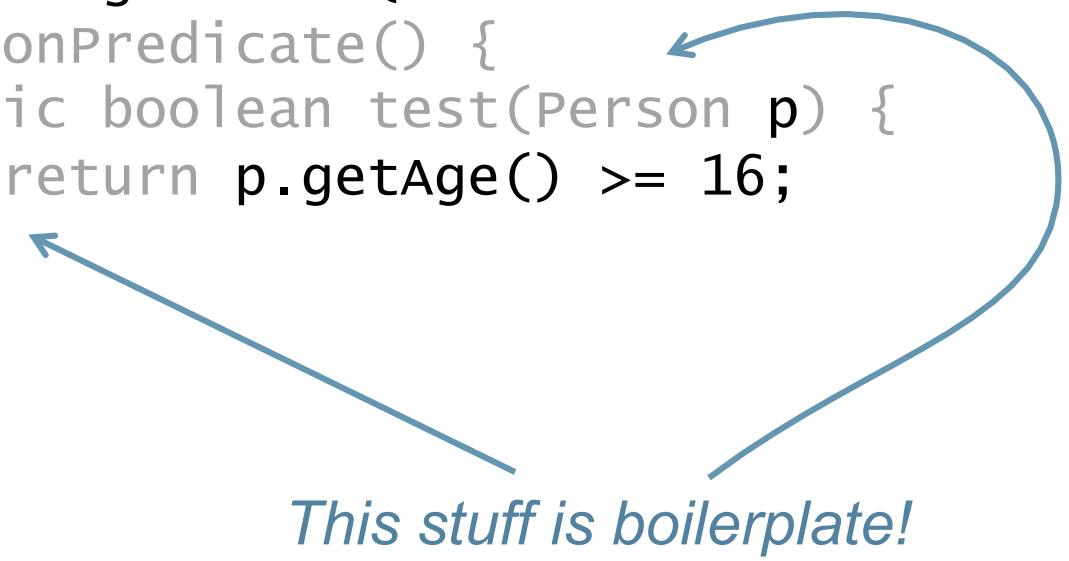
Let's Look At The Code Again

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

This is the significant part.

Let's Look At The Code Again

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



Let's Look At The Code Again

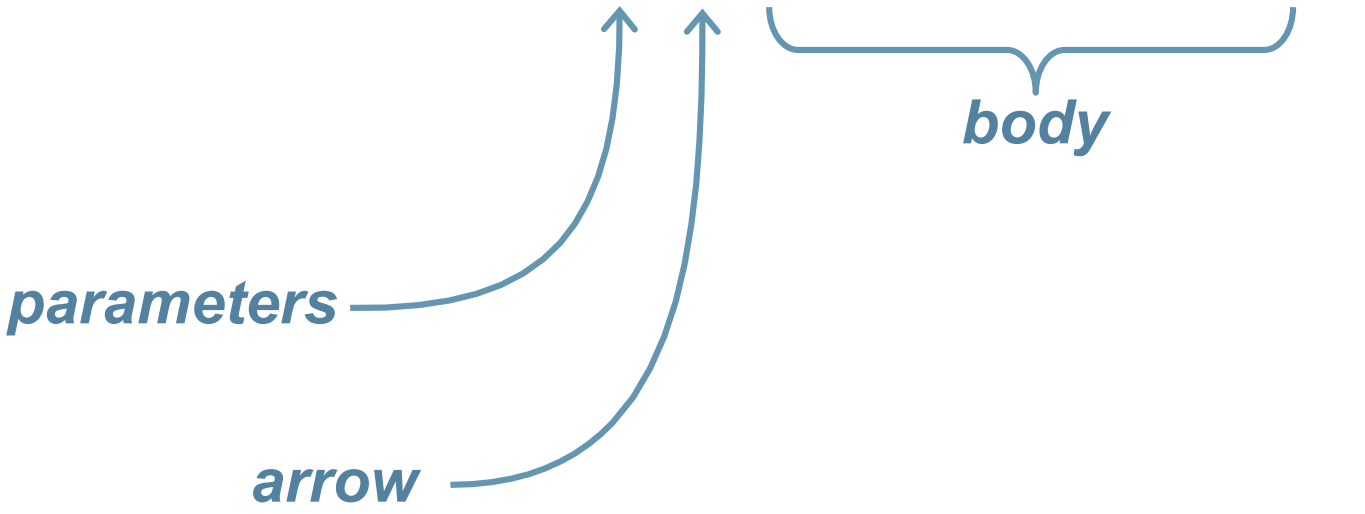
```
void robocallEligibleDrivers() {
    robocallMatchingPersons(

                                          p
                        -> p.getAge() >= 16


    );
}
```

Erase the boilerplate!

Collapse whitespace.

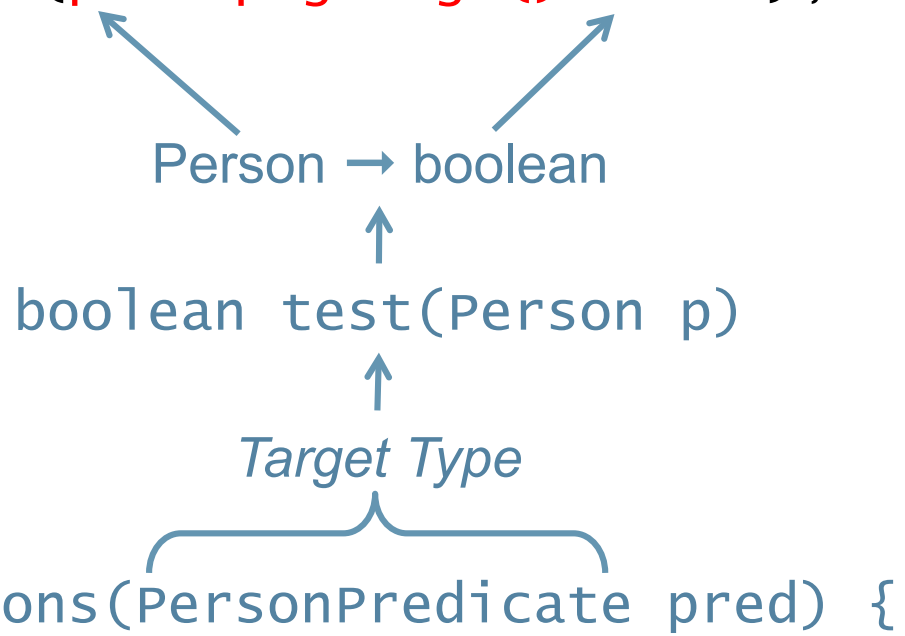
Our First Lambda Expression!

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



What Happened to the Types?

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



They're still there, they're just inferred

Let's Look At The Code Again

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

The lambda is sort-of an abbreviation for this.
(But it's not implemented that way.)

Rewrite Using Lambda

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

Let's Look At The Code Again

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

Rewrite Using Lambda

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

Back to the Library...

```
void robocallMatchingPersons(PersonPredicate pred) {
    for (Person p : list) {
        if (pred.test(p)) {
            PhoneNumber num = p.getPhoneNumber();
            robocall(num);
        }
    }
}
```

In the library, a lambda looks like an instance of an interface, and you just call a method on it. The library can't tell whether it was passed a lambda or an anonymous inner class.

Notes on Lambda Syntax

```
(int a, int b) -> a + b

(int a) -> a + 1
```

expression lambdas

```
() -> 42

(int a, int b) -> { println(a + b); return a + b; }

(int a) -> { println(a + 1); return a + 1; }

() -> { println("Returning 42."); return 42; }
```

Notes on Lambda Syntax

```
(int a, int b) -> a + b

(int a) -> a + 1

() -> 42
```

statement lambdas

```
(int a, int b) -> { println(a + b); return a + b; }

(int a) -> { println(a + 1); return a + 1; }

() -> { println("Returning 42."); return 42; }
```

Notes on Lambda Syntax

```
(a, b) -> a + b

(a) -> a + 1

() -> 42

(a, b) -> { println(a + b); return a + b; }

(a) -> { println(a + 1); return a + 1; }

() -> { println("Returning 42."); return 42; }
```

Types may be omitted if they can be inferred from the target type

Notes on Lambda Syntax

```
(a, b) -> a + b

a -> a + 1

() -> 42

(a, b) -> { println(a + b); return a + b; }

a -> { println(a + 1); return a + 1; }

() -> { println("Returning 42."); return 42; }
```

Parentheses may be omitted if the lambda has a single parameter

Default Methods

- New language feature to support ***interface evolution***
- Prior to Java SE 8, adding a method to an interface was incompatible
  - existing implementations would be missing that method
  - callers would get AbstractMethodError
- Default methods allow a method declaration ***and implementation*** to be added to interfaces

Default Methods

- Implementing classes can override default methods with specialized implementations
- Implementing classes lacking override inherit default methods
- Callers cannot distinguish default methods from ordinary methods
- Default methods not directly related to lambda
  - many default method APIs added in support of lambda

Default Method Syntax

```
interface Map<K,V> {
    V get(Object key);
    V put(K key, V value);
    // ...

    default V getOrDefault(Object key, V defValue) {
        V v = get(key);
        if ((v != null) || containsKey(key))
            return v;
        else
            return defValue;
}
```

New APIs Using Lambda

```
Iterable.forEach(lambda)

Collection.removeIf(lambda)

List.replaceAll(lambda)

List.sort(lambda)

Map.computeIfAbsent(key, lambda)

Map.merge(key, value, lambda)
```

New APIs Using Lambda

```
List<Person> list = ... ;


for (Person p : list) {
    System.out.println(p);
}


list.forEach(p -> System.out.println(p));
```

New APIs Using Lambda

```
List<Person> list = Collections.synchronizedList(...);


for (Person p : list) {
    System.out.println(p);
}


list.forEach(p -> System.out.println(p));
```

Default Methods Added to Collections

```
Collections.sort(list,
    (p1, p2) -> p1.getName().compareTo(p2.getName())
);
```

*Collections.sort has one algorithm that must work with **all** List implementations.*

```
list.sort(
    (p1, p2) -> p1.getName().compareTo(p2.getName()));
```

The List.sort default method can be overridden by List subclasses to provide an optimized implementation.

Summary

- Lambda: anonymous functions or function literals
  - functions that can be passed or returned as data
  - enable parameterization of behavior
- They work with other Java 8 language changes
  - default methods on interfaces
  - enhanced type inference in the compiler
- New APIs added in support of Lambda
  - various default methods added throughout libraries
  - new Streams API: aggregate operations, parallelism

Java 8 Resources

- Download: java.oracle.com
- Documentation & Tutorials: docs.oracle.com/javase
- Source code: openjdk.java.net/projects/jdk8
- FAQ: lambdafaq.org
- Technical documents: openjdk.java.net/projects/lambda
  - *State of the Lambda*
  - *State of the Lambda: Libraries Edition*

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.


Java™