Lambda Expression

First-class Citizens in Java

- Whole point of a programming language is to manipulate values, which are called **first-class values or citizens**.
  - int, double, reference type are first-class citizens in Java.
- Other structures in our programming languages, which can not be passed around during program execution.
  - method, class are second-class citizens in Java.
- Could we make it possible **to pass a method as a citizen at runtime**?
- Java 8 designers added the ability to express this directly in Java.
  - lambda expression (anonymous method)
  - method reference

Why lambdas for Java?

- Provide libraries a path to multicore
  - Parallel-friendly APIs need internal iteration
- Empower library developers
  - Enable a higher degree of cooperation between libraries and client code
- It's about time!
  - Java is the lone holdout among mainstream OO languages at this point to not have closures
- Inner classes give us some of these benefits, but are to clunky
- How to represent lambda expressions at runtime is not a trivial question

Lambda expressions for Java

- What is the type of a lambda expression?
  - Most lanuages with lambdas have some notion of a function type
  - Java has no concept of function type
- Adding function types would create many questions
  - How do we represent functions in VM type signatues?
  - How do we create instances of function-typed variables?
- **Want to avoid significant VM changes**
  - Java Designers have historically modeled functions using single-method interfaces such as Comparator (compare())
  - Rather than complicate the type system, let's just formalize that
    - Give them a name: **"Functional Interfaces"**
    - Always convert lambda expression to instance of a functional interface
- Lambda is just an inner class instance
  - This means (among other things) one class per lambda expression
  - Performance issues

Main Scenario

- Suppose that you are creating a social networking application.

- You want to create a feature that enables an administrator to perform any kind of action, such as sending a message, on members of the social networking application that satisfy certain criteria.
  - Administrator specifies criteria of members on which to perform a certain action.
  - Administrator specifies an action to perform on those selected members.
  - Administrator selects the Submit button.
  - The system finds all members that match the specified criteria.
  - The system performs the specified action on all matching members.

class Person

- Suppose that members of this social networking application are represented by the following Person class:

```
public class Person {
    public enum Gender { MALE, FEMALE }

    String name;
    LocalDate birthday;
    Gender gender;
    String emailAddress;

    public int getAge() { // ... }
    public void printPerson() { // ... }
}
```

Search for Members

- Suppose that the members of your social networking application are stored in a List<Person> instance.

```
public static void printPersonsOlderThan(List<Person> roster, int age) {
    for (Person p : roster) {
        if (p.getAge() >= age)
            p.printPerson();
    }
}
```

- This approach can potentially make your application brittle
  - The class records and measures ages with a different data type or algorithm
  - What if you wanted to print members younger than a certain age
  - You may need each method searches for members that match one characteristic, such as gender, name

Search Criteria Code in an interface

- To specify the search criteria, you implement the CheckPerson interface.
- You can use an **anonymous class** too.

```
interface CheckPerson {
    public abstract boolean test (Person p);
}

public static void printPersons(
    List<Person> roster, CheckPerson tester) {
    for (Person p : roster) {
        if (tester.test(p))
            p.printPerson();
    }
}
```

```
class CheckPersonEligibleForSelectiveService
                                implements CheckPerson {
    public boolean test (Person p) {
        return p.gender == Person.Gender.MALE &&
            p.getAge() >= 18 &&
            p.getAge() <= 25;
    }
}
printPersons(  roster,
        new CheckPersonEligibleForSelectiveService());
```

Functional Interface

- A functional interface is any interface that contains only one abstract method.
- The CheckPerson interface is a functional interface.

```
@FunctionalInterface
interface CheckPerson {
    public abstract boolean test (Person p);
}
```

- A functional interface may contain one or more default methods or static methods.

Search Criteria Code with a Lambda Expression

- Instead of using an anonymous class expression, you use a lambda expression:
  - @FunctionalInterface: error message could be "Multiple non-overriding abstract methods found in interface CheckPerson" if you define more than one abstract method

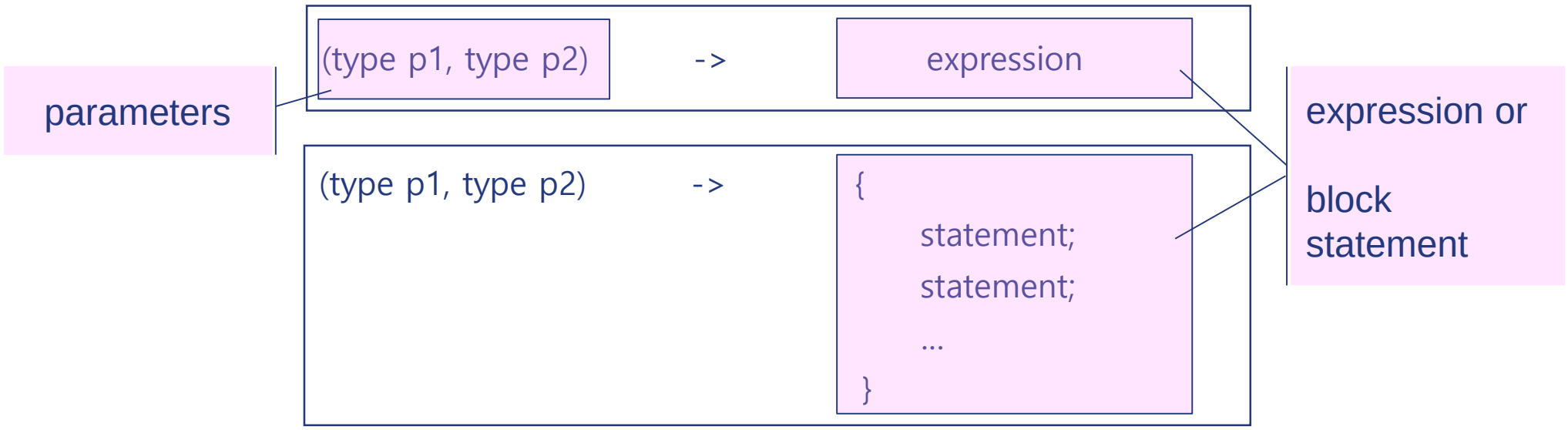
```
@FunctionalInterface
interface CheckPerson {
    public abstract boolean test(Person p);
}
public static void printPersons(
    List<Person> roster, CheckPerson tester) {
    for (Person p : roster) {
        if (tester.test(p))
            p.printPerson();
    }
}
```

```
//lambda expression syntax
// (parameters) -> expression
printPersons( roster,
                (Person p) -> p.getAge() >= 18
              );

printPersons( roster,
    (p) -> p.getGender() == Person.Gender.MALE
            && p.getAge() >= 18
            && p.getAge() <= 25
          );
```

Syntax of Lambda Expressions

- A comma-separated list of formal parameters enclosed in parentheses.
  - You can omit the data type of the parameters in a lambda expression
  - you can omit the parentheses if there is only one parameter
- The arrow token, ->
- A body, which consists of a single expression or a statement block.



Lambda Parameters

- The parameters of a lambda expression have to match the parameters of the method on the single method interface

```
printPersons( roster, (Person p) -> p.getAge() >= 18);
```

```
@FunctionalInterface
interface CheckPerson {
    public abstract boolean test (Person p);
}
```

Lambda Parameters

- If the **parameter types can be inferred**, you can omit them.

```
printPersons( roster, (p) -> p.getAge() >= 18);
```

Lambda Parameters

- Zero Parameters

```
() -> System.out.println("Zero parameter lambda");
```

- One Parameter

```
(param) -> System.out.println("One parameter: " + param);
param -> System.out.println("One parameter: " + param);
```

- Multiple Parameters

```
(p1, p2) -> System.out.println("Multiple parameters: " + p1 + ", " + p2);
```

- Parameter Types

```
(Car car) -> System.out.println("The car is: " + car.getName());
```

Lambda Function Body

- The body of a lambda expression is specified to the right of the -> in the lambda declaration

```
(oldState, newState) -> System.out.println("State changed")
```

- If your lambda expression needs to consist of multiple lines, you can enclose the lambda function body inside the { } bracket

```
(oldState, newState) -> {
  System.out.println("Old state: " + oldState);
  System.out.println("New state: " + newState);
 }
```

Returning a Value

- You can return values from Java lambda expressions
- You just add a return statement to the lambda function body

```
(param) -> {
  System.out.println("param: " + param);
  return "return value";
}
```

- In case all your lambda expression is doing is to calculate a return value and return it, you can specify the return value in a shorter way

```
(a1, a2) -> { return a1 > a2; }

(a1, a2) -> a1 > a2
```

Lambda Expression Examples

Use case	Examples of lambdas
A boolean expression	`(List<String> list) -> list.isEmpty()`
Creating objects	`() -> new Apple(10)`

Consuming from an object	`(Apple a) -> {`
	`  System.out.println(a.getWeight());`
	`}`

Select/extract from an object	`(String s) -> s.length()`
Combine two values	`(int a, int b) -> a * b`
Compare two objects	`(Apple a1, Apple a2) ->` `a1.getWeight().compareTo(a2.getWeight())`

Modern Java in Action: Lambda, streams, functional and reactive programming, 2018

Package java.util.function

- Functional interfaces provide target types for lambda expressions and method references.

Interface	Description	Method
Predicate<T>	Represents a predicate (boolean-valued function) of one argument.	boolean test(T t)
Function<T,R>	Represents a function that accepts one argument and produces a result.	R apply(T t)
Consumer<T>	Represents an operation that accepts a single input argument and returns no result.	void accept(T t)
Supplier<T>	Represents a supplier of results.	T get()
UnaryOperator<T>	Represents an operation on a single operand that produces a result of the same type as its operand.	extends Function<T,T>
BinaryOperator<T>	Represents an operation upon two operands of the same type, producing a result of the same type as the operands.	extends BiFunction<T,T,T>
BiFunction<T,U,R>	Represents a function that accepts two arguments and produces a result.	R apply(T t, U u)

https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html

Examples

Use case	Example of lambda	Matching functional interface
A boolean expression	`(List<String> list) -> list.isEmpty()`	`Predicate<List<String>>`
Creating objects	`() -> new Apple(10)`	`Supplier<Apple>`
Consuming from an object	`(Apple a) -> System.out.println(a.getWeight())`	`Consumer<Apple>`
Select/extract from an object	`(String s) -> s.length()`	`Function<String, Integer>` or `ToIntFunction<String>`
Combine two values	`(int a, int b) -> a * b`	`IntBinaryOperator`
Compare two objects	`(Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight())`	`Comparator<Apple>` or `BiFunction<Apple, Apple, Integer>` or `ToIntBiFunction<Apple, Apple>`

Modern Java in Action: Lambda, streams, functional and reactive programming, 2018

Predicate<T>

- You might use this interface when you need to define a lambda that represents **a predicate (boolean-valued function)** of one argument.

```
public static void printPersonsWithPredicate(
    List<Person> roster, Predicate<Person> tester) {
    for (Person p : roster) {
        if (tester.test(p)) {
            p.printPerson();
        }
    }
}

printPersonsWithPredicate(
    roster,
    p -> p.getGender() == Person.Sex.MALE
                && p.getAge() >= 18
                && p.getAge() <= 25
);
```

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

Function<T, R>

- You might use this interface when you need to define a lambda that maps information from an input object to an output

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.function.Function;
public class FunctionalIntefaceExample {
  public static void main(String[] args) {
    List<Integer> list = map(
        Arrays.asList("lambdas", "in", "action"), (String s) -> s.length() );
    System.out.println(list); // [7, 2, 6]
  }
  public static <T, R> List<R> map(List<T> list, Function<T, R> f) {
    List<R> result = new ArrayList<>();
    for ( T t: list ) result.add(f.apply(t));
    return result;
  }
}
```

```
@FunctionalInterface
public interface Function<T, R> {
  R apply(T t);
}
```

Target Typing

- The type expected for the lambda expression inside the context is called the **target type**.
  - The type of a lambda is deduced from the context in which the lambda is used.
  - You can only use lambda expressions in situations in which the Java compiler can determine a target type (e.g., method arguments)

```
printPersonsWithPredicate(roster, (Person p) -> p.getAge() >= 18);
```

- The type-checking process is deconstructed as follows:
  - First, you look up the declaration of the printPersonsWithPredicate method.
  - Second, it expects, as the second formal parameter, an object of type Predicate<Person> (the target type).
  - Third, Predicate<Person> is a functional interface defining a single abstract method called test.
  - Fourth, the test method takes an Person and returns a boolean

Same Lambda, Different Functional Interface

- The same lambda expression can be associated with different functional interfaces if they have a compatible abstract method signature.

```
//Assignments context

Supplier<Integer> c1 = () -> 42;

Callable<Integer> c2 = () -> 42;
```

```
public interface Supplier<T> {
    T get();
}
public interface Callable<V> {
    V call();
}
```

- Which method will be invoked in the following statement?

```
String s = invoke(() -> "done");
```

```
void invoke(Runnable r) {
    r.run();
}

<T> T invoke(Callable<T> c) {
    return c.call();
}
```

  - The method invoke(Callable<T>) will be invoked because that method returns a value

Variable Capture

- A Java lambda expression is capable of accessing variables declared outside the lambda function

```
public interface MyFactory {
   public String create(String message);
}
```

```
public class LocalVariableCapture {
   public static void main(String[] args) {
      String greeting = "Hello";
      MyFactory myFactory = (message) -> {
         return greeting + ":" + message;
      };
      System.out.println(myFactory.create("Java Lambda"));
   }
}
```

- Java lambdas can capture the following types of variables:
  - Local variables (**only effective final** is allowed because it is on the stack)
  - Instance variables in the class
  - Static variables in the class

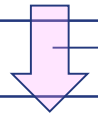
Method References (::)

- Method references as syntactic sugar for lambdas that refer only to a single method because you write less to express the same thing.

```
Person[] rosterAsArray = roster.toArray(new Person[roster.size()]);
public class Person {
    public static int compareByAge (Person a, Person b) {
        return a.birthday.compareTo(b.birthday);
    }
}
```

```
// The method signature of sort in Arrays class
static <T> void sort ( T[] a, Comparator<? super T> c )
```

```
Arrays.sort(rosterAsArray, (a, b) -> Person.compareByAge(a, b) );
```



method reference (syntactic sugar)

```
Arrays.sort(rosterAsArray, Person::compareByAge );
```

- Each has the following characteristics:
  - Its formal parameter list is copied from Comparator<Person>.compare, which is (Person, Person).
  - Its body calls the method Person.compareByAge.

https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html

Example of Method Reference

- Method references can be seen as shorthand for lambdas calling only a specific method.

Lambda	Method reference equivalent
`(Apple apple) -> apple.getWeight()`	`Apple::getWeight`
`() -> Thread.currentThread().dumpStack()`	`Thread.currentThread()::dumpStack`
`(str, i) -> str.substring(i)`	`String::substring`
`(String s) -> System.out.println(s)`	`System.out::println`
`(String s) -> this.isValidName(s)`	`this::isValidName`

Kinds of Method References

- There are four kinds of method references:

Kind	Syntax	Examples
static method	ContainingClass::staticMethodName	Person::compareByAge Integer::parseInt
instance method of a particular object	containingObject::instanceMethodName	myVariable::compareByName
instance method of a particular type	ContainingType::methodName	String::length
constructor	ClassName::new	HashSet::new

Practice – Sorting a List

```
List<Person> rosterAsList =
                Arrays.asList(new Person("Lee"), new Person("Park"), new Person("Kim"));

class PersonNameComparator implements Comparator<Person> {
    public int compare(Person a, Person b) {
        return a.getName().compareTo(b.getName());
    }
}
rosterAsList.sort( new PersonNameComparator() );
```

first-class

```
// The method signature of sort in List class
default void sort(Comparator<? super E> c)
```

```
rosterAsList.sort( new Comparator<Person>() {
    public int compare(Person a, Person b) {
        return a.getName().compareTo(b.getName());
    }
} );
```

anonymous class

```
rosterAsList.sort( (a, b) -> a.getName().compareTo( b.getName() ) );
```

lambda

```
rosterAsList.sort( Person::compareByName );
```

method reference

Q&A